NAVAL POSTGRADUATE SCHOOL
Monterey, California

THESIS

A BIOLOGICALLY BASED APPROACH TO THE MUTATION OF CODE

by

Loretta L. Vandenberg

September 1999

Thesis Advisor: Taylor Kidd

Approved for public release; distribution is unlimited.

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

1. AGENCY USE ONLY *(Leave blank)*	2. REPORT DATE September 1999	3. REPORT TYPE AND DATES COVERED Master's Thesis

TITLE AND SUBTITLE : A Biologically Based Approach to the Mutation of Code	5. FUNDING NUMBERS
6. AUTHOR(S) Vandenberg, Loretta L.	
7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) Naval Postgraduate School Monterey, CA 93943-5000	8. PERFORMING ORGANIZATION REPORT NUMBER
9. SPONSORING / MONITORING AGENCY NAME(S) AND ADDRESS(ES) N/A	10. SPONSORING / MONITORING AGENCY REPORT NUMBER

11. SUPPLEMENTARY NOTES

The views expressed in this thesis are those of the author and do not reflect the official policy or position of the Department of Defense or the U.S. Government.

12a. DISTRIBUTION / AVAILABILITY STATEMENT Approved for public release; distribution is unlimited.	12b. DISTRIBUTION CODE

13. ABSTRACT *(maximum 200 words)*

Evolutionary programming is a relatively new problem solving approach in the field of computer science. It attempts to model the processes of natural selection and evolution to solve complex problems. This technique is very powerful because it can be applied to a wide range of problems, and can find solutions that other more traditional techniques cannot.

This research attempts to augment the methodology of an evolutionary programming approach with two new features: (1) dominant and recessive traits and (2) intron and exon regions. These features form the basis of a specialized approach for evolutionary programming which might be able to be applied to new problem areas where evolutionary programming usually performs poorly.

This specialized approach is applied to the well known problem of a series expansion, so that the results are easily compared to a known solution, and that the influence of these additional mechanisms on the population of solutions can be studied. Results from implementing the new mechanisms individually and together are presented and compared with a baseline evolutionary programming implementation.

14. SUBJECT TERMS Genetic Programming, Computing and Software	15. NUMBER OF PAGES 167
	16. PRICE CODE

17. SECURITY CLASSIFICATION OF REPORT Unclassified	18. SECURITY CLASSIFICATION OF THIS PAGE Unclassified	19. SECURITY CLASSIFICATION OF ABSTRACT Unclassified	20. LIMITATION OF ABSTRACT UL

NSN 7540-01-280-5500

Standard Form 298 (Rev. 2-89)
Prescribed by ANSI Std.

239-18

Approved for public release; distribution is unlimited

A BIOLOGICALLY BASED APPROACH TO THE MUTATION OF CODE

Loretta L. Vandenberg
Captain, United States Marine Corps
B.S., Washington and Lee University, 1990

Submitted in partial fulfillment of the
requirements for the degree of

MASTER OF SCIENCE IN COMPUTER SCIENCE

from the

NAVAL POSTGRADUATE SCHOOL
September 1999

ABSTRACT

Evolutionary programming is a relatively new problem solving approach in the field of computer science. It attempts to model the processes of natural selection and evolution to solve complex problems. This technique is very powerful because it can be applied to a wide range of problems, and can find solutions that other more traditional techniques cannot.

This research attempts to augment the methodology of an evolutionary programming approach with two new features: (1) dominant and recessive traits and (2) intron and exon regions. These features form the basis of a specialized approach for evolutionary programming which might be able to be applied to new problem areas where evolutionary programming usually performs poorly.

This specialized approach is applied to the well known problem of a series expansion, so that the results are easily compared to a known solution, and that the influence of these additional mechanisms on the population of solutions can be studied. Results from implementing the new mechanisms individually and together are presented, and compared with a baseline evolutionary programming implementation.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

ACKNOWLEDGEMENT

The author would like to acknowledge and thank those individuals who provided their support throughout the “evolution” (every pun intended) of this thesis…

To my family, especially my Dad, without whom nothing would be worth it.

To my guardian angels, Mom and Doc, without whom I wouldn’t have made it. I miss you both.

To Prof Kidd for having a very weird idea.

To Prof Rasmussen for saving my life.

To Shirley Oliveira for all the candy and hugs.

To Shirley Kidd, Steve Balmer, and Mike Saluto for not letting me get fat.

To Dave and Katrina Hensley for always letting me know what really is important and never letting me forget it.

To Mike Saluto for making think it was worth it. It is. I hope you realize it someday. Don’t forget to tell the swans about me.

To my kitties, Bruiser and Bandit, for constantly interrupting me with their circuit training over my keyboard, eating my stickies, and for rendering the cutest baffled looks when scolded at which I could never stay angry very long.

And finally, to Tracy Braun for liking the lime, laser pointer tag, “ocean adventures,” and letting me push his buttons. I owe you more than you could ever know. “Face it, Tiger, you hit the jackpot.”

--Rett

I. INTRODUCTION

A. PURPOSE

The idea of imbuing computers with life-like features, such as intelligence, is not a new idea. Early computer scientists, like Alan Turing, asked such questions as, "Can machines think?" He never did answer that question, but he did propose that an intelligent machine would have to be a learning machine and that evolution could be used as an intelligent learning process [Ref 1]. As a result of his work, and that of other pioneering computer scientists, the link between computation and "life" found a firm foundation from which sprouted the fields of Artificial Intelligence (AI) and Evolutionary Computation (EC).

There are many arguments for what the real definition and underlying purpose of "evolution" are, but one common theme among the arguments is that evolution is responsible for adaptations. In trying to develop models for how adaptation occurs in natural organisms and importing those models to computers, John Holland created Genetic Algorithms (GAs). Since their introduction in the 1960's, GAs have been applied to solve pressing computational problems because they can be used to search for solutions to a specific problem among an intractable number of possibilities [Ref 2].

Genetic Programming (GP) is one of several techniques stemming from GAs. While GP uses GAs in its implementation, GP differs from a GA because it does not attempt to solve a problem directly. Instead of building an evolutionary program to solve the problem, GP uses GAs to automatically generate programs that will help the computer to solve a problem without specifically being programmed.

This thesis will describe and attempt to implement a specialized GA using a GP approach. This specialized GA incorporates certain specific genetic characteristics and mechanisms that occur in complex biological organisms in an effort to create a self-mutating algorithm that can be applied to adaptive applications.

B. RESEARCH QUESTIONS

It is necessary to clearly establish which underlying biological/genetic mechanisms are used in this specialized GA and enumerate the benefits gained by choosing those mechanisms. The general characteristics of these mechanisms must be described unambiguously within an algorithm in order for those characteristics to map to a computer based analogue population.

Assuming that a specialized GA has been developed, it is necessary to determine whether or not the application of this specialized GA achieves results comparable to traditional GAs when applied to similar problems. If the new approach works, but is preemptively slow to complete, than its usefulness, outside of being a novel approach, is limited. However, if this specialized GA can solve problems that traditional GAs cannot, then it could maintain legitimacy as a computationally viable problem solving method.

Because this new approach is a GA, this thesis addresses two fundamental questions pertinent to all GAs: Which initial population size and probability of mutation should be used, and how many generations must be run before reliable results are produced? The GP portion of the implementation also requires that the problem to be solved be broken down into its appropriate functions and terminals.

C. THESIS OUTLINE

Upon completion of the introduction, but prior to answering the research questions posed, it is necessary to thoroughly explain some fundamental concepts in molecular genetics and evolutionary computation. These explanations are presented in Chapter II and are needed to understand the details of the computational analogues this thesis develops. Additionally, there has been work completed in related fields that has had either direct or indirect influence on the formation of this new idea. These seminal works are discussed and summarized in detail in Chapter III. The combination of the background information from Chapter II and the overview of previous works from Chapter III set the foundation for describing why this thesis' research questions are pertinent and how they will be answered (Chapters IV and V, respectively).

The experiment was run on a modified version of the GPSYS (pronounced "gipsys") software [Ref 3]. The modifications made are covered in Chapter V. A statistical analysis was done on the results, the outcome of which is given in Chapter VI. Chapter VII summarizes the important contributions of this thesis.

D. EXPECTED BENEFITS OF THIS THESIS

If it is possible to exhibit evolutionary/adaptive traits by using this specialized GA, then it would seem possible to address a larger spectrum of problems than is possible with traditional genetic algorithms. If this specialized GA works without loss of optimization, then it is possible that techniques discovered during this research could be used to develop adaptive applications in the future. If a larger class of problems can be addressed, the genetic algorithms resulting from this work might be successfully applied where such were not previously possible. The benefits of such self-adapting algorithms to the DoD are obvious and widespread.

II. BACKGROUND

A. AN OVERVIEW OF MOLECULAR GENETICS

In 1839, two German microscopists, Matthias Schleiden and Theodor Schwann postulated that all organisms were constructed from fundamental units called cells, and that all cells arise from other cells. Without having witnessed the mechanism of mitosis, or being able to see inside the nucleus, they accurately determined that the cell is the fundamental unit of all organisms [Ref 4].

In 1858, Charles Darwin and Alfred Wallace published their theory proposing that evolution in organisms occur because of natural selection. They stated that various forms of life are not constant but are continually giving rise to slightly different forms, some of which are adapted to survive and multiply more effectively. At that time, they did not know the origin of this continuous variation, but they realized that these new characteristics had to persist in progeny if such variations were to form the basis of evolution [Ref 5].

In 1865, Gregor Mendel laid the precursor to the rules of heredity when he postulated that various traits are controlled by pairs of factors. Three years later, Ernest Haeckel postulated that the nucleus is responsible for heredity. It was not until the early 1900's and the work of Hugo De Vries that all three biologists were proven correct when the pairs of chromosomes found in the cell nucleus were discovered to be the active factors responsible for heredity. Although Darwin and Wallace's theory of evolution was to become orthodox among biologists by the late 1800's, it was not until the acceptance of genetic theory that evolution theory would be embraced by the general world and other scientific communities [Ref 6].

In 1868, a Swiss biologist named Friedrich Miescher first identified deoxyribonucleic acid (DNA). It was not until 1944 with the work of Oswald Avery, Colin MacLeod, and Maclyn McCarty that it was believed that the DNA found within the genes was responsible for the physical transmission of hereditary traits from generation to generation. Alfred Hershey and Martha Chases proved DNA was the actual genetic material in 1952. Since then, subsequent advances in molecular genetics, population genetics, microscopy and molecular biology have allowed scientists to better hypothesize how organisms are able to evolve [Ref 7].

1. A Molecular Genetics Primer

The following discussion of molecular genetics utilizes terminology familiar to anyone knowledgeable of fundamental cell biology. Appendix A provides a brief review of cell biology for the reader who needs to familiarize himself with these terms.

In order to be absolutely clear about the genetic mechanisms that are being mapped to a computational analogue, it is necessary to ensure that there is a common, accepted understanding of what these mechanisms are and how they work. It is also necessary to introduce some genetic terminology that may not be familiar to most readers. Therefore, this section describes the basic structures and functions of genes.

Every living organism is created from a complete set of instructions called a **genome**. This genome contains the master blueprint for all cellular structures and functions within the organism for the duration of its life. The genome is defined by the DNA, which when combined with protein molecules, called histones, form **chromatin fiber**. These fibers form structures known as **chromosomes**, which are found in the nucleus of all non-somatic cells.

Many **procaryotic** organisms have single-stranded DNA. An organism with a single DNA chain is called **haploid**. Many unicellar and all higher order organisms are **eucaryotic** and have double-stranded, or double helical as it is more popularly known, DNA. Organisms with double-stranded DNA are called **diploid**.

Each DNA molecule contains many **genes**, which are a specific sequence of nucleotide bases found at a particular position, or **locus**, on the chromosome[1]. These genes hold all the information necessary for constructing proteins and enzymes for the biochemical reactions that are necessary to maintain life[2]. DNA, however, does not actually produce proteins. A single-stranded copy of the gene, called ribonucleic acid (RNA), is created for that purpose in a process called **transcription**. This process is similar to replication, except that an exact double-stranded copy of the entire DNA molecule is not the end product. The RNA molecule is not only single-stranded, but also

[1] There are different types of DNA found in a cell, such as nuclear and mitochondrial DNA. They both are the templates for RNA and protein production, thus the exact differences will not be discussed. All references to DNA will refer to nuclear DNA, although, in general, the mechanisms apply to both types.

[2] It is more correct to say that each gene holds the relationships for a specific polypeptide. Often proteins are made from several polypeptide chains, each of which is the product of a separate gene.

significantly shorter in length because it is a copy of only a portion of the DNA. This RNA molecule leaves the nucleus and is used to create proteins in the cell's ribosomes[3]. The RNA is interpreted and the protein is made by a process called **translation**.

DNA → RNA → protein

Transcription Translation

Figure 1. The Flow of the Genetic Code

Not all sequences in DNA are eventually transcribed into RNA. In fact, the entire DNA sequence of eucaryotic organisms is a mixture of these transcribed and untranscribed regions. These untranslated, intervening DNA sequences are called **introns**. The functional areas of DNA from which the RNA is transcribed are called **exons**. Genes whose DNA is a complex mixture of introns and exons are referred to as being **split** or **discontinuous**.

Proteins, made in the translation process, are polymers of amino acids, of which there are twenty used in the synthesis of proteins (see Appendix B). The ribosomes are able to retrieve the correct amino acid because each amino acid is encoded by a certain sequence of nucleotide bases in the RNA strand, which is then placed in the protein being constructed. This sequence consists of three bases and it is called a **codon**. Each codon codes for a specific amino acid. Different codons that code for the same amino acid are called **synonyms** (see Figure 2). What is commonly referred to as the "genetic code" is the relation between the codons and the amino acids which they represent. This code is nearly, but not absolutely, universal.

Alleles are alternative forms of a gene that can occupy a particular chromosomal site. Whereas genes code a trait, such as eye color, alleles are different "settings" for the trait, such as blue or hazel eyes[4]. A gene is "expressed" when the information coded within it is converted into structures present and operating within the cell. Whether expressed or not, the particular set of genes contained in the organism's genome is called

[3] There are several types of RNA. See Appendix B for clarification.

[4] More precisely, the locus refers to the place on the chromosome where an allele resides. An allele is just the bit of DNA at that place. A locus is a template for an allele. An allele is an instantiation of a locus.

the **genotype**. The genotype gives rise to the **phenotype**, which is the aggregate of physical and mental characteristics of the organism, such as eye color, height, etc.

SECOND POSITION

FIRST POSITION	U	C	A	G	THIRD POSITION
U	phenyl-alanine	serine	tyrosine	cysteine	U
					C
	leucine		stop	stop	A
			stop .	tryptophan	G
C	leucine	proline	histidine	arginine	U
					C
			glutamine		A
					G
A	isoleucine	threonine	asparagine	serine	U
					C
			lysine	arginine	A
	* methionine				G
G	valine	alanine	aspartic acid	glycine	U
					C
			glutamic acid		A
					G

* and start

Figure 2. The Genetic Code

Genes are arranged in a fixed, linear order. This order can change, but this is rare. . Movable DNA segments, called **transposons,** occassionally jump around chromosomes, thus fundamentally altering the chromosomal structure. In addition to neatly moving genes, transposons also scramble DNA, making deletions, inversions, and other rearrangements. It is becoming clear that such changes are a critical feature of chromosomal evolution. Because recombination and transposition generate new combinations of genes, they enlarge the repertoire from which natural selection chooses. Gene rearrangements also regulate DNA expression because gene location and orientation may determine whether a gene is silent or active.

2. Population Genetics

Population genetics is the scientific discipline that is concerned with the genetic basis of evolution. It studies the frequencies and fitness of genotypes in natural populations. Evolution is the change in the frequencies of genotypes through time, with the change possibly due to differences in fitness. Changes in genotype frequencies, though, are not easily measured, because the time scale associated with the introduction of most naturally occurring genetic variants is very long and are thus impossible to directly observe. Mathematical models of evolution are used instead, and the behavior of the model is compared to that of the natural population.

One important effect that population genetics has explained is that of dominant alleles within the genotype on the evolution of the organism. The type of dominance the allele exhibits determines how it affects the overall genotype and whether or not the allele will be maintained in the population. Allele dominance explains why some mutations can be sustained within the population, while others are not. Appendix C explains some of these theories in more detail.

3. Some Biological Causes of Evolution

The fundamental causes of life and its subsequent evolution are still a matter of speculation. Scientists still cannot answer the questions of why life formed or what caused species to differentiate. They can compare DNA or protein sequences and apply elements of statistical geometry to determine when the species diverged. They can offer statistical arguments as to what happened to cause the changes, but the exact causes are still unknown. Therefore, the following discussion is only a theory and, as with any theory outside of mathematics, is potentially wrong. However, it is also emphasized that this following theory is not trying to prove the causes of evolution in vivo, but to find useful evolutionary and genetic mechanisms for which computational analogues can be made.

Most genes of higher eucaryotes have many introns. Lower eucaryotes have a much higher proportion of continuous genes. Comparisons of the DNA sequences of genes encoding proteins that are highly conserved in evolution suggest that introns were present in ancestral genes and were lost during the evolution of organisms that have become optimized for very rapid growth, such as bacteria. The presence of introns has had an evolutionary effect, especially with regard to the development of complex organisms [Ref 8].

Exons encode discrete functional units of proteins or can encode for the whole protein. The former idea has been proven, which leads to the attractive hypothesis that new proteins arose in evolution by the rearrangement of exons [Ref 9]. Shuffling exons is a rapid and efficient means of generating novel genes because it preserves functional units while allowing them to interact in new ways. Introns are regions where DNA can break and recombine with no deleterious effects. Therefore, the presence of introns increases genetic variation by allowing more discontinuity. The greater the discontinuity of the DNA, the more exons are found in the DNA, and thus more possible combinations of exons.

Since the developmental potential of an organism is determined by its genes, DNA must necessarily mutate as organisms evolve. But evolutionary changes occur only rarely. Since living cells require the correct functioning of thousands of proteins, each of which could be damaged by a mutation at many different sites in the cell's gene, it is clear that DNA sequences almost always are passed on unchanged if progeny are to have a good chance of survival.

Naturally occurring mutations include almost all conceivable changes in DNA sequences (see Appendix B). Mutations that have only a subtle effect on a gene product, such as temperature-sensitive mutations, are often the result of a simple switch of one base for another. However, there are natural mutations that destroy the function of a gene completely. These more drastic changes, called **null mutations**, include not only base switches, the insertions and deletions of a base, but also extensive insertions and deletions and even gross rearrangements of chromosome structure. Such changes might be caused, for example, by the insertion of a transposon, which typically places many thousands of bases of foreign DNA in the coding sequence of a gene, or by an aberrant cellular recombination process.

One way to reduce the chances of a harmful mutation occuring is to increase the area in which a mutation can occur that will either have no effect or at least not harm the organism[5]. Introns are usually extensive in length and account for about half of the total

[5] Correction enzymes ensure that replication and recombination occur without error. Excluding radical modifications of DNA brought about by outside factors such as radiation, DNA has a built-in repair mechanisms.

DNA molecule. Having such a large portion of inactive regions decreases the chance of a harmful mutation occurring in an exon[6]. Thus, the introns give additional protection from harmful mutations.

Recombination is not accidental, but is instead an essential cellular process catalyzed by enzymes which are made by the cell's ribosomes. Besides providing genetic variation, recombination enzymes allow cells to retrieve sequences lost when DNA is damaged. By switching specific segments within chromosomes, cells put dormant genes into sites where they can be expressed, even creating new protein-coding regions (see Appendix C).

Synonyms in the genetic code demonstrate the idea of degeneracy. If more than one codon did not code for the same amino acid, then only twenty codons would designate amino acids and the rest would be stop signals. The probability of mutating to a chain termination signal would then be much greater, and chain-termination mutations usually lead to inactive proteins. Substitutions of one amino acid for another are usually relatively harmless. It also allows for the DNA base composition to vary over a wide range without altering the amino acid sequence of the proteins encoded by the DNA. Therefore, degeneracy allows mutations to occur while increasing the probability that the mutation will not be deleterious. It also increases the variation of the genotype without destroying necessary functions.

Double stranded DNA, as opposed to single stranded, allows for more variation in an individual's genome, and hence the entire population. On a single stranded gene, there might be n different alleles at a single locus, so the individuals in that population can have n different genotypes resulting from differences at that locus. In a double stranded gene, there are n homozygous combinations and $n(n-1)/2$ non-repetitive heterozygous combinations[7].

Organisms with double stranded DNA also follow Mendel's law of dominance[8]. Dominant and recessive features provide an interesting dynamic within the population's

[6] This assumption is true if every nucleotide or base pair in the strand had equal probability of mutation. There are examples of regions where mutation is more likely to occur (see Appendix B), but in the highly accurate replication process, all points have an equal probability.

[7] This is not taking into account combinations that would lead to non-viable individuals and so would never naturally occur.

[8] This is a drastic simplification. Not all alleles are entirely dominant or entirely recessive. See Appendix C for further explanation.

genotype, allowing for increased complexity. The idea of dominant and recessive traits is important in evolution because it often determines the viability of the individual in the population and the environment. Those individuals exhibiting dominant traits are usually said to be better adapted to the environment and thus are more likely to survive. However, having dominance in one area does not mean the individual is the most fit in the population, nor is it a guarantee that offspring will retain the trait.

In summary, introns, degeneracy within the genetic code, double-stranded DNA, and dominance have all played a role in evolution. They have either acted to reduce the deleterious effects of mutation or have acted to increase the complexity of DNA. These in turn affect the possibilities of how a species can change over generations.

4. Biological Basis of Evolutionary Computation

Because evolution is, in effect, a method of searching among an enormous number of possibilities for a "solution," it has inspired researchers trying to solve computational problems. In biology, the large number of possibilities is the set of possible genetic sequences, and the desired "solutions" are highly fit organisms. A highly fit organism is one capable of surviving and reproducing in its environment. For those computational problems that require programs to be adaptive—i.e., continuing to perform well in a changing environment—evolution is capable of searching a constantly changing set of possibilities. Furthermore, evolution is a massively parallel search method: rather than working on one species at a time, evolution tests and changes millions of species in parallel.

EC, which will be more thoroughly discussed in the next section, is the blanket term used to describe the class of computer-based problem solving systems that use computational models of known mechanisms of natural evolution as key elements in their designs and implementations. All the systems simulate the evolution of individual structures, within a population of structures, via the processes of selection, mutation and recombination. Each individual in the population has some type of genetic material and each receives a measure of its fitness in the environment based on some type of evalution of that genetic material.

From the previous section, one can see that biologists have identified many principles, from the genetic level and higher, which govern the evolution of living things. At the highest level, the theory of natural selection governs the evolutionary adaption of the biological world. Natural selection operates on the organism through its performance on one specific task—the production of offspring. In Darwin's words, it is "survival of the fittest." Those individuals that are fittest in the environment are more likely to survive, and thus propagate their genetic material. Individuals which reproduce are chosen in the selection process. In the computational world, there are several ways to accomplish selection, but usually the fitness is considered.

During the reproduction process, assuming reproduction is not asexual, some genetic material is taken from one parent, and some from the other. Recombination can occur either during gamete formation, from which the offspring will be subsequently formed, or can occur directly to create the new individual. In EC, recombination is better known as crossover, although even in biology, these terms are synonymous (see Appendix B).

Computational mutation is generally done during the crossover procedure. It can occur during gamete formation, the actual crossover, or both. It is generally a stochasitic process, with each locus in the genetic material having a certain probability of being mutated during the procedure. Mutations which occur outside the process of "being born" are called "cosmic ray" mutations.

In summary, EC techniques maintain a population of structures, called individuals, that evolve according to the rules of selection, recombination and mutation specified by the user. Each individual in the population receives a measure of its fitness in the environment. Reproduction usually focuses on highly fit individuals, thus using their fitness information to keep their genetic material within the population. Recombination and mutation, which perturb the genetic material during reproduction in order to create new individuals, is a means to explore the fitness landscape and find more solutions.

B. INTRODUCTION TO EVOLUTIONARY COMPUTATION

There are several main methods in the evolutionary computation field, distinguished mainly by the types of structures that comprise the individuals in the population. These differences determine the factors by which one individual may differ from another, and thus the allowable genetic variation. Equally important differences also exist in the genetic operators used to create offspring, as well as many of the selection procedures based on fitness and other parameters. The predominant styles are Evolution Strategies (ES), Evolutionary Programming (EP), Genetic Algorithms (GA), and Genetic Programming (GP).

These different styles, while representing true differences in approach, were each developed by different and initially unrelated groups of people with little cross-fertilization of ideas in the early days of development. It is a commonly held opinion that even though more individuals have applied more than one of these approaches to solving problems, the number of people who could be considered truly interdisciplinary is small. The relationships and individual strengths and weaknesses of each of these styles are just beginning to be understood [Ref 10].

ESs are frequently associated with engineering optimization problems. The structures that undergo adaptation are typically sets of physically measurable objective variables that are associated with similarly measurable strategy variables in an individual. Fitness is determined by executing task specific routines and algorithms using objective variables as parameters. Strategy variables control the way in which mutation varies each objective variable during the production of new individuals. Recombination is usually applied to both objective variables and strategy variables.

EP operates on a variety of representational structures, frequently real-valued objective variables or finite state machines. The objective variables are arguments to task specific routines and algorithms designed to solve a specific problem. Mutation is the only genetic operator employed, with significant strategy built into the overall algorithm to direct the mutation in a computationally beneficial direction.

GAs usually operate on fixed length character strings, often binary, as the structure undergoing adaptation. Other representational structures are possible. Fitness is determined by executing task specific routines and algorithms using an interpretation of the character string as their parameters. Crossover is the principal genetic operator employed, with mutation usually included as an operator of secondary importance.

GP is an offshoot of Genetic Algorithms in which the computer structures that undergo adaptation are themselves computer programs. Specialized genetic operators are used which generalize crossover and mutation for the computer programs undergoing adaptation. Generally, the programs being evolved are represented as trees.

C. GENETIC PROGRAMMING FUNDAMENTALS

John Koza developed GP in 1992, although he originally applied GAs to Lisp expressions. Other languages have since implemented GP techniques, but they require that the program be represented in a hierarchical structure (i.e., tree). Regardless of the implementation, Koza says that there are six steps to solving a problem using GP. These steps are choosing (1) the terminals, (2) the functions, (3) the control parameters, (4) the termination criteria, (5) the fitness function, and determining (6) the program's architecture. The functions are the internal nodes of the tree, while the terminals are the leaves. The control parameters are similar to those in GAs: the population size, the number of generations, and the probabilities of crossover and mutation. The termination criteria is usually reached when a program is found that solves the given problem, or after a certain number of generations. The fitness function is similar to that used in GAs. The last step refers to defining the number of automatically defined functions (ADFs) [Ref 11].

Along with these six steps, Koza also defined the necessary conditions for terminals and functions. These conditions must be chosen so that they are able to express a solution to a given problem. This property is known as sufficiency. The second property is that of closure, which is satisfied when each of the functions can accept as its arguments any value or data type that might be returned by any function or be taken on by any terminal. As long as this property is fulfilled, crossover is possible and the offspring tree will be syntactically correct and executable. One method of maintaining closure is to use a Strongly Typed Genetic Program (STGP). In a STGP, variables, constants, arguments, and returned values can be of any data type with the provision that the data type for each value is specified beforehand [Ref 12].

The use of ADFs in GP induce a divide-and-conquer strategy in which the problem to be solved is decomposed into smaller sub-problems. The subproblems are usually easier to solve and the results can be combined together to build the solution for the original problem. In the case of the subproblems, their solutions are usually subroutines of the whole program. In GP, these subroutines can be obtained by using special subtrees having a fixed number of branches. This subtree can then be encapsulated to act as a terminal for another program, because it returns a value that is used by that node in evaluating its function.

III. OVERVIEW OF PREVIOUS WORK

A. THE COMPILING GENETIC PROGRAMMING SYSTEM (CGPS)

Most GP approaches use an interpreter to execute the programs that are developed from the provided problem specific language. The code segments in the population are decoded at runtime by a virtual machine. However, interpreted code usually causes a large overhead and executes much slower than compiled code. In 1993, Peter Nordin created the Compiling Genetic Programming System (CGPS), which used the machine's code as the programs in the population and eliminated the interpreter [Ref 13]. He used the CGPS to make a classifier function that differentiates nouns and non-nouns of Swedish words strictly by spelling. He compared the execution of the CGPS with that of a combined neural network and interpreting GP system applied to the same problem. The CGPS performed significantly better. The CGPS is now commercially available through AIM Learning Systems™ as the Discipulus™ and Discipulus Pro™ programs, which are GP/Simulated Annealing programs used to conduct computerized automatic learning and learning/optimization tasks at the machine code level[9].

One other significant feature of Nordin's work is that he used the C language. The individuals in the population are machine code sequences resembling a standard C-function (see Figure 3). Valid C-functions are put together at runtime directly in memory by a GA.

The header in the function call gets its arguments from the stack. The footer cleans up after the function is completed. The same header and footer are added at the beginning of the initialization of each individual in the population. The mutation and crossover operations are prevented from modifying these sections.

The return instruction forces the system to leave the function and return control to the calling procedure. The placement of the return instruction is allowed to vary if the system is implemented on a Complex Instruction Set Computer (CISC), because CISC computers have variable length instructions. The return instruction is given a range in which it can be placed if run on that system. The CGPS, however, was tested on a limited Sun-4 instruction set with programs of fixed length.

[9] The author did not accertain what the differences are between the CGPS and the Discipulus™ products.

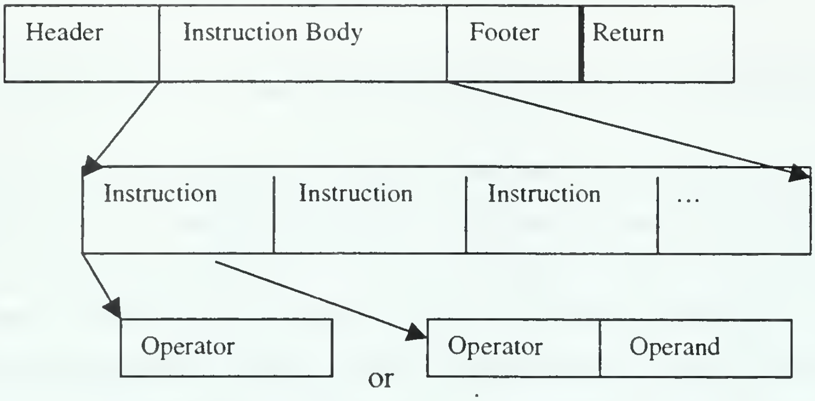


Figure 3. Generic Machine Code Function Structure

The instruction body is the actual program that evaluates the function. Two classes of instructions are used. In the first class, instructions are performed between registers or as unary functions. Instructions in the second class require a constant operand to immediately follow the operator. The CPGS only uses two-register Sun-4 instructions limited to two addressing mode types; a total of twenty-four instructions were used.

The individuals in the population are randomly generated and run through a GA using steady-state tournament selection and a standard uniform crossover. Mutation can work on the operator or, if it exists, the operand. If the operator is mutated, it can only be changed to a member of the set of approved instructions. No jumps, illegal instructions or loops are allowed.

For his experiments, the population size was varied between 20 and 4,000 individuals, with 40,000 individuals total being processed. The typical size of individuals tested was 7, 12 and 28 instructions. The population was allowed to run through 21 training sets. He concluded that his population was able to learn the training set about as quickly as the neural network[10]. The benefit of the CPGS in this instance is that it only took one minute to train as compared to the neural network's 235 minutes. Individuals in the CPGS also only needed approximately 50 bytes of memory as opposed to the neural

[10] CPGS learned 86% of the words as compared to 89% for the neural network. On unknown words, CPGS classified 72% correctly compared to the neural network's 69%.

network's 450 bytes. He also concluded that smaller populations worked better than larger ones.

B. THE GENETIC EVOLUTION OF MACHINE LANGUAGE SOFTWARE SYSTEM (GEMS)

In 1995, Ronald L. Crepeau developed Genetic Evolution of Machine Language Software (GEMS) in which he used a large set of generalized operators and instructions vice problem specific ones. It was thought that a large generalized set of operators and terminals would make the GP process inefficient. He wanted to determine if that assumption was true[11] [Ref 14]. Unlike Nordin, though, his system did not go on to a commercial implementation.

GEMS consisted of three parts: the microprocessor emulator, a pool of machine language (ML) programs, and the Genetic Process Controller (the GPC). Like Nordin, he decided to work at the machine level in order to preclude the need to compile. He wrote an emulator for the Z80™ architecture, which is a 16-bit microprocessor with a large instruction set, 64 Kilobyte addressing range, and input-output features. Included in the emulation were seven, 8-bit wide registers, some of which could be combined to form 16-bit registers, as well as Stack Pointer (SP), Program Counter (PC), and Flag (F) registers.

The individual instructions of pool members, i.e., the ML programs mentioned above, were implemented using a C structure which defined the number of bytes in the instruction and an array that held the actual instruction. A pool member's set of instructions were stored as an array of these structures. Memory contents were stored within the pool member itself, with program memory and data memory segregated.

The GPC generated new pool members, linked them for execution within the emulator, evaluated the fitness of pool members, and controlled the breeding, mutation and survival of pool members. Pool members were pair-wise bred and could be bred one or more times. A double crossover was used with preference given to the most fit of the parents. Only one child was produced per crossover. Each offspring was evaluated, and if it was found to be more fit than either of its parents, it subsequently replaced the weaker parent in the pool. If it was not more fit than its parents, it did not survive.

[11] He also wanted to see the effects of adding memory.

There were two types of mutations in GEMS. The first type of mutation replaced a random amount of contiguous program and memory values in an offspring prior to its fitness evaluation. The other type randomly and completely replaced a weak pool member.

GEMS was applied was to the problem of generating an ML agent that in one run would output the string "Hello World". The fitness of the individual was based on outputting the correct string. Additional value was added if the ML agent could output the correct string in the shortest amount of instructions.

Three benchmarks were tested for each run. Crepeau measured the number of generations it took until a pool member first output the correct string, how many generations passed until a correctly performing ML agent was stable within the pool, and when the shortest agent in the pool was less than 100 instructions. Runs were not normally terminated until all three benchmarks were reached, but there were cases when ML agents of less than 100 instructions were not produced.

Pool size varied between 150 and 2000 members, each of which were run sixteen times. For populations of less than 500 individuals, there was a significant standard deviation, so for this particular problem, a larger population performed better. One interesting conclusion he found was that the complexity of the problem appears to increase linearly, vice combinatorially or exponentially when increasing the length of the output string. Although there are specific issues resulting from the GEMS implementation and the problem on which it was used, he proved that "agents of simple functionality can be generated with a GP process that involves a large number of ML operators and memory as implemented in GEMS" [Ref 15].

C. DIPLOIDY/DOMINANCE IN GENETIC SEARCH

Although not the first example of utilizing diploidy and dominance in a GA approach, in [Ref 16], F. Green considers the diploid chromosome as two chromosomes with one gene each. He used a 0-1 knapsack problem in order to see how the GA performed on a problem with a changing global optima. He speculated that if a similar environment repeatedly arose, diploidy could enable the species to rapidly re-express its former fitness for that environment.

Individuals of the diploid population were represented as a C++ object based on the Genitor GA [Ref 17]. The Genitor GA originally only worked on a haploid chromosome but was modified so that the initial population produced would replicate the values from the first homologue into the second. Member functions for the class were added to handle diploid fitness evaluation.

He based his dominance relationship on the enzyme production of some genes in biological systems. A homozygous dominant and the heterozygote both exhibit the dominant feature, but the homozygote actually produces more of the enzyme, specifically, twice as much as the heterozygote. In his organisms, allele values are the fitness values of the two strings on each gene in the corresponding chromosomes. These fitness values represent the intermediate, or the enzyme, produced. These values compete prior to producing the observed phenotype. Dominance is implemented by mapping the two intermediate, or sub-phenotype, fitness values to the scalar fitness value that will subsequently be used for selection. The mapping function, referred to as the dominance function, simply chooses the maximum of the two values.

D. EXPLICITLY DEFINED INTRONS AND DESTRUCTIVE CROSSOVER

In [Ref 18], Peter Nordin, Frank Francone and Wolfgang Banzhaf investigated introns in GP populations. Introns in GP are nothing more than evolved code fragments that do not effect the fitness of the individual. They divided the introns into two categories, Explicitly Defined Introns (EDI) and Implicit Introns (II). IIs develop through the process of evolution. EDIs were specifically added into the program structures, which were then run through the CGPS mentioned previously. An EDI is a structure that does not affect the fitness calculation of the individual, but does affect the probability of crossover between adjacent blocks of evolved code. IIs are introns that emerge from the code itself.

They identified many classes of code segments that behaved as introns, but chose to evaluate code segments (1) that did not contribute to the fitness, and (2) had the property where each node could be replaced by a no operation (noop) instruction without affecting the output for any of the fitness cases. The number of introns in an individual is counted by first making the noop substitution and then counting the number of times there is no affect on the fitness. The effective length of the individual is the absolute length of the individual's chromosome minus the number of introns.

They discovered that both types of introns protect an entire individual or code block against the effects of destructive crossover. EDIs and IIs were also found to work together. EDIs helped to keep IIs around until the perfect individual was found, at which time their existence dropped to nearly zero. That dropping off may be a good indication as to when to stop training the population.

IV. FORMAL DEFINITION OF THE PROBLEM

Evolutionary biology and genetics are very complex subjects; it is necessary to understand each part of a highly specialized, complicated system and how the parts interact with each other to affect the whole. There are many questions biological/genetic researchers have yet to adequately answer as to why things are the way they are and how they work. Because this thesis is an attempt to coordinate the unrelated disciplines of computer science and biology/genetics, it is crucial that the genetic mechanisms have been either proven or have been adequately theorized before they are chosen. If the mechanisms chosen are not a cause of evolutionary changes in organisms, then the approach of this thesis is faulty. If the mechanisms cannot be clearly defined, then it will be difficult, if not impossible, to unambiguously express their functionality as an algorithm that can be implemented on a computer.

Of the three previously stated genetic mechanisms—degeneracy, intron/exon regions, and dominant/recessive traits—only the latter two will be implemented. Although these mechanisms have been implemented individually in previously discussed research, there has been no research uncovered that combine the two. The mechanism required to implement degeneracy is very complex and outside the scope of this thesis. To demonstrate that the specialized GA works, it will be applied to a problem to which a traditional genetic algorithm/program has been used successfully (i.e., to find the answer to a series expansion). If it can be proven that this specialized GA can evolve a working program, the next question to be answered is whether or not it can do better than, the same as, or worse than a traditional GA/GP performance.

As covered in Chapter I, this is a new GA approach. From Chapter II, though, there are many variations of GAs and other EC methods, and it is arguable that this new approach is actually a GP because of its implementation. Each of these categories uses different characteristics and different parameters. Experimental results will help to better classify as to where this new GA best fits, but even without knowing in advance what characteristics and parameters will be most important, there are general questions that need to be answered.

As mentioned in Chapter I, the three basic questions of what initial population size to use, what probability of mutation to use, and how many generations must be produced before reliable results are produced, must be addressed[12]. Controlled runs, where only one of these variables is changed, are done in order to establish a working baseline. These variables are simply provided to the program prior to the start of the run. Initial population sizes of 20, 50, 100, 500, 1000 and 5000 individuals are commonly used throughout the literature and so are used here. The probability of mutation most used in the literature is 0.001. As this experiment is particularly interested in the effects of mutation, it is necessary to increase mutation's occurrence. Values of 0.01, 0.1 and 1 are used[13]. With regards to the number of generations, typically between 500 and 1000 generations has proven to be enough. While conducting the control experiments, though, some runs were be allowed to reach 10,000. This series should give enough data such that the minimum generation number required for reliable results can be determined[14].

One important question with respect to GAs and GP is that of the representation of individuals. Representation in GAs is characterized by the chromosome, whereas in GP, representation is characterized by the functions, terminals and data on which they operate. In this paradigm, there are two representations, the genotype and the phenotype, with the genotype actually being a program. The phenotype is the result that is produced by "running" the genotype. Since the implementation is done using a GP tool, the genotype is represented by a parse tree holding randomly chosen values from a function and terminal set. The GP tool's code has been modified to define introns (other than those implicitly produced) and dominant/recessive genes. Each individual's genotype is randomly generated using the specialized GA so that there are individuals expressing recessive traits. Chapter V is a description of the system and the modifications that were made.

Another important aspect of GAs and GP is that of choosing and testing the fitness function. Fitness functions usually define boundary conditions, but how to best

[12] Population size is intimately tied to the problem being solved. Each problem, though, has a minimum necessary size in order to reliably find a solution. As it is yet unclear whether or not this specialized GA works, population sizes that are common in the literature are used.

[13] 0.001% mutation is much larger than that which occurs naturally. The goal is to demonstrate the function of the intron regions. As such, the percentages must be higher in order to achieve some observable difference within a reasonable number of generations.

[14] Allowing the population to run indefinitely will continue to result in changes. Similar numbers to more traditional GAs are used in order to make the results from the new GA being explored in this thesis more compatible with the new GA's older counterpart.

choose a fitness function is not the primary focus of this thesis[15]. The fitness function is important because it determines the genetic program's ability to evolve. The focus of the thesis, though, is to see the effect of incorporating dominant/recessive pairs and introns within the structure of an evolving program. Because this research desires to use this specialized GA to evolve a program, this GA must be applied to a problem for which the sufficiency property can be fulfilled (see Chapter II, Section C). Regardless of the problem being solved, one measure of fitness is whether or not an individual finds the correct answer. Additionally, partial credit needs to be given to individuals who were close to finding the answer. Extra credit should be assigned to individuals who found the answer most efficiently or in the fewest number of instructions executed. This implies that unless the value of the output is null, every individual starts with a value from which points would be subtracted or added depending on the fitness evaluation prior to the awarding of extra credit.

This new GA is used to complete a series expansion. This problem involves a comparison to a known solution. The series expansion can be evaluated for a given value of x by expanding the series for a certain number of terms. An individual's fitness can be determined by the absolute difference between the "real" answer (i.e., one computed on a calculator) and the individual's return value. Since there is a minimum required number of operations to be performed in a series expansion, those individuals who can find a good solution in the least number of operations should receive a higher fitness than individuals who find the same answer in more operations.

In order to determine whether or not the specialized GA works, it is necessary to be able to determine the average fitness of the population, and the fitness and structure of the most fit individual over the variety of control parameters (e.g., population size, generations and probability of mutation). In the case of the series expansion problem, the absolute difference between the real answer and the population/best individual answer should be very small if the individual's fitness is to be considered a good one.

Once the introns are incorporated into the program, not only is it necessary to track the average fitness and the fitness and structure of the best individual, but also the

[15] One of the original inspirations for this thesis was the idea of increasing the search space by allowing "dead" individuals. A "dead" individual is defined as one with low or no fitness value assigned to it. This would prevent it from being selected to procreate in a Darwinian world. The "dead" individuals would be allowed to remain within the population and would be subject to genetic operations. Although that idea did not come to fruition, it is still intriguing.

number of introns in the best-fit and worst-fit individuals, as well as the average number of introns per individual in the population. If introns are beneficial in finding better solutions, then the best individual should have more of them, most of the population will have some of them, and the worst will have few, if any.

Dominance should be implemented separately from introns and tested before the two are combined within the environment. In order to determine the effect of dominance on the population, it is necessary to determine the number of dominant and recessive genes in the best-fit and worst-fit individuals, as well as the average number of dominants and recessives within the population.

When the two are combined, all of the above data must be tracked. To determine if they work, each stage must be compared to the original test runs and compared to the previous test results.

In summary, the fundamental question this thesis attempts to answer is whether or not this specialized GA works, meaning whether this specialized GA succeeds in creating a population of individuals that is capable of finding a program that, when executed, is the solution to the problem. Assuming this specialized GA is shown to work, the next step is determine under what parameters does it perform best.

V. DESCRIPTION OF THE SYSTEM AND METHODS USED TO DETERMINE EVOLUTION

A. GPSYS-1.1

GPSYS-1.1 was written by Adil Qureshi in 1997 at the Department of Computer Science, University College, London, United Kingdom [Ref 19]. The primary reason for choosing it over others is because it is written in Java, which is better understood by this author than C++ implemented tools such as GP-Quick [Ref 20], GALib [Ref 21], lilgp [Ref 22], Genitor [Ref 23], Avida [Ref 24] and GPCPP [Ref 25]. GPSYS is a Strongly Typed GP, with many built-in primitives, generic functions and terminals. It includes commented source code along with javadoc documentation for all classes. Example problems are provided with GPSYS which were invaluable in learning how to use the system.

B. STRUCTURE OF GPSYS-1.1

GPSYS uses a tournament selection to choose the individuals who will be mutated, crossed over and replaced within the population. In GPSYS, a default value of seven individuals are choosen at random from the population to compete in the tournament[16]. The number of individuals competing in each tournament is assigned by the *tournamentSize* variable defined in the *GPParameters* class. This class must be compiled before beginning a run in order for a new tournament size to be applied. The *GPParameters* class is also where the probability of mutation, number of generations to evolve and population size are defined.

The *Population* class holds the array of *Individuals* for each generation that will undergo the operations of selection, crossover and mutation. The *Individual* class is the evolved GP program. The *Population* class has the evolve function as well as the selection methods for tournaments, mutation and crossover. When the population evolves, new individuals are created either by mutation or by crossover, but not both. The probabilities are based on a random seed provided at the start of the run. This random seed is defined by Java's *Random* class.

[16] Selection is random and is not influenced by an individual's fitness or any other attribute.

Every *Individual* contains a *Chromosome* class and is assigned a fitness value based on the *Fitness* class. The *Chromosome* class represents the tree whose nodes are members of the *Gene* class. The *Chromosome* class also defines the genetic operators. If a mutation is chosen as the evolving mechanism, a tournament is run from which is selected the most and least fit individuals. The most fit individual will be cloned and the clone will be mutated by selecting a branch of the tree at random and replacing the branch with a new branch. The new branch is generated in one of two ways. One method is to generate a tree of the maximum specified depth, which is defined in the *GPParamters* class. The second method builds a tree in which the probability of encountering a leaf is the same as that of an internal node. Which method is used is probabalistically determined and based on the random seed. This mutated clone will then replace the least fit individual selected by the tournament within the population.

If crossover is the chosen evolutionary mechanism, then a tournament is run, from which three individuals are selected: the most fit (mother), the next best-fit (father), and the worst fit. Incestuous crossovers are permitted, although a warning is given should such occur[17]. The mother is cloned and a branch from the clone is randomly selected for replacement. A branch from the father is chosen, copied, and placed into the clone, creating the child[18]. If the child's depth exceeds the maximum as a result of the crossover, it is returned to its original state as simply a clone of its mother. The worst-fit individual selected in the tournament is then replaced by the child.

The *Gene* class is a node in the GP tree. Since GP trees are based on functions and terminals, the *GeneTerminal* and *GeneFunction* classes extend the Gene class. Both classes use the *Primitives* class, which represents a unit in the evolved program. This class is extended by the *Terminal* and *Function* classes, where *Terminal* defines standard Java primitives such as integer, float, and object. The *Functions* class can be extended in order to implement user-defined functions. GPSYS comes with many predefined functions such as arithmetic and logical operations.

[17] An incestuous crossover was never incountered during these runs, but there is nothing provided within the system to prevent it from happening.

[18] The branch chosen from the father must return the same type as that selected from the clone for replacement.

C. HOW TO INCORPORATE INTRONS INTO GPSYS

Because GPSYS-1.1 is fully object oriented, it is extensible, which is a very desirable feature for making changes. One very big problem with a Java implementation is the lack of pointers, making it difficult to assign the null value[19]. The first change made by this research was to extend the *Primitive* class, which defined the fundamental types used within the system. The *Primitive* class is actually generic, with extensions defining the *Terminals* and *Functions* that can be used without the user having to define his own. The *Function* class was modified by the inclusion of a no-operation (noop) instruction. This instruction returns no values, takes no arguments, and does nothing. This function will be used at the root node of an intron. It can be found within the subtree of the intron, but the highest node connecting the intron to the parse tree must be a noop. Luckily, the *Type* class, which defined all the off-the-shelf usable types, also included a no argument constructor defined as "No Type."

The *Gene* class was modified with a boolean flag indicating whether or not the gene is an intron. All terminals will set this flag to false because terminals are the values used by the functions. The *Gene* class is generic and is extended by the *GeneTerminal* and *GeneFunction* classes. A no-operation class called *Nop* extends the *Function* class. The user can ensure introns are made by including *Nop* as one of the available functions when setting the GP parameters (*GPParameter* class) before compiling and running the population.

The *Population* class had to be modified in order to collect the necessary data on the entire population (average number of introns per individual) as well as to retain the fitness, number of introns, and structure of the best and worst individuals.

D. HOW TO INCORPORATE DOMINANT TRAITS INTO GPSYS

Implementing dominant/recessive traits required modifying many of the same classes. The first changes made were to extend as many of the predefined functions as possible. The predefined functions are the dominants. These functions, which are all classes, are extended to create the recessives. If one is present, the recessive functions all subtract a value away from the individual's fitness.

[19] Only a Java *Object* can be assigned a null value. Primitive types cannot be assigned to null, nor can primitives be type casted to an *Object* which could then subsequently be set to null.

The *Gene* class is also modified to include a boolean flag which is true if the gene is dominant, false otherwise[20]. An additional allele flag is included that allows the gene to be identified as homozygous dominant, homozygous recessive or heterozygous. During crossover, this flag is checked in both parents to ensure the offspring produced is a Mendelian possibility with the probability of creation of that type equal to the Mendelian probability based on the law of dominance (see Appendix C).

Just as with the introns, the *Population* class had to be further modified in order to be able to determine the average number of dominant and recessive traits in the population, as well as for the best-fit and worst-fit individuals.

C. ANALYSIS STRATEGY

To recapitulate from Chapter IV and summarize the above discussion, the following tables are provided to graphically depict the experiments run and the data collected. On the test runs, which performed a series expansion without any modification to GPSYS, the following data was collected.

Probability of Mutation	Values tested were 0.0, 0.001, 0.01, 0.1, 0.5, 1.0
Number of Generations[21]	Values tested were 10, 50, 100, 250, 500, 1000, 5000, 10000
Population Size	Values tested were 20, 50, 100, 500, 1000, 5000, 10000
Average Fitness	Evaluated by the *Population* class
Average Complexity	Complexity refers to the number of nodes in the evolved tree
Best-Fit Fitness	The best answer in the population
Best-Fit Complexity	For a comparison to the average complexity

Table 1. Test Run Parameters Used and Data Produced

[20] Terminals cannot be dominant or recessive since they are acted on, and so the flag is set to false for all terminals.

[21] GPSYS tended to crash while running the larger populations for a longer number of generations. Data is continuously written to file as the run progresses, so some data is saved even in the event of failure. Because of the tendency to fail, tests on larger populations (10,000 or 5,000 individual population) run for longer generations were either not conducted, or the data used is that which was collected up the time of the failure.

The fitness values were determined by the following functions:

Series Expansion: $x^n + x^{n-1} + \ldots + 1$, where n is random, but does not exceed 10, $0.0<x<25$

Fitness Function:

| eqn above – individual's return value|

Table 2. Fitness Functions for Test Run

Once the introns were added, the following data was necessary:

Probability of Mutation	Values tested were 0.0, 0.001, 0.01, 0.1, 0.5, 1.0
Number of Generations	Values tested were 1000
Population Size	Values tested were 500, 1000
Average Fitness	Evaluated by the *Population* class
Average Complexity	Complexity refers to the number of nodes in the evolved tree
Best-Fit Fitness	The best answer in the population
Best-Fit Complexity	For a comparison to the average complexity
Worst-Fit Fitness	The worst answer in the population
Worst-Fit Complexity	For a comparison to the average complexity
Best-Fit Number of Introns	For a comparison to the average population and worst
Worst-Fit Number of Introns	For a comparison to the average population and best
Average Number of Introns	Evaluated by the *Population* class

Table 3. Parameters Used and Data Produced for Tests after Introns are Included

The dominant/recessive genes required the following data:

Probability of Mutation	Values tested were 0.0, 0.001, 0.01, 0.1, 0.5, 1.0
Number of Generations	Values tested were 1000
Population Size	Values tested were 1000
Average Fitness	Evaluated by the *Population* class
Average Complexity	Complexity refers to the number of nodes in the evolved tree
Best-Fit Fitness	The best answer in the population
Best-Fit Complexity	For a comparison to the average complexity
Worst-Fit Fitness	The worst answer in the population
Worst-Fit Complexity	For a comparison to the average complexity
Best-Fit Number of Recessive Genes	For comparison to the average/worst dominants
Best-Fit Number of Dominant Genes	For comparison to the average/worst dominants
Worst-Fit Number of Recessive Genes	For a comparison to the average population and best
Worst-Fit Number of Dominant Genes	For comparison to the average/best dominants
Average Number of Recessive Genes	Evaluated by the *Population* class
Average Number of Dominant Genes	Evaluated by the *Population* class

Table 4. Parameters Used and Data Produced for the Evaluation of Dominant/Recessive Pairs

The combination of the two required the following data to be collected:

Probability of Mutation	Values tested were 0.0, 0:001, 0.01, 0.1, 0.5, 1.0
Number of Generations	Values tested were 1000
Population Size	Values tested were 1000
Average Fitness	Evaluated by the *Population* class
Average Complexity	Complexity refers to the number of nodes in the evolved tree
Best-Fit Fitness	The best answer in the population
Best-Fit Complexity	For a comparison to the average complexity
Worst-Fit Fitness	The worst answer in the population
Worst-Fit Complexity	For a comparison to the average complexity
Best-Fit Number of Recessive Genes	For comparison to the average/worst dominants
Best-Fit Number of Dominant Genes	For comparison to the average/worst dominants
Best-Fit Number of Introns	For a comparison to the average population and worst
Worst-Fit Number of Recessive Genes	For a comparison to the average population and best
Worst-Fit Number of Dominant Genes	For comparison to the average/best dominants
Worst-Fit Number of Introns	For a comparison to the average population and best
Average Number of Recessive Genes	Evaluated by the *Population* class
Average Number of Dominant Genes	Evaluated by the *Population* class
Average Number of Introns	Evaluated by the *Population* class

Table 5. Parameters Used and Data Produced for Dominant/Recessive and Intron Runs

VI. RESULTS

A. GEOMETRIC SERIES EXPANSION CONTROL EXPERIMENTS

The purpose behind conducting this first set of control experiments was to determine system behavior and with which population size and number of generations the system performed best. The judging criterion used to determine best performance is based on finding the fittest individual within the shortest number of generations. The results of these runs were subsequently used to determine the population size and number of generations to be used with the introns and dominant/recessive genes experiments.

From Chapter IV, the population sizes varied between 20 and 10,000, with the number of generations running to 10,000. The probabilities of mutation used were selected from 0.001, 0.01, 0.1 and 1.0. Each run used the same random seed in order to determine what effect population size and probability of mutation had on the overall run. All tests were conducted on a Dell^{TM} Dimension XPS D300 with a Pentium II processor running Microsoft Windows95^{TM} as the operating system. All code was compiled with the Java Development Kit version 1.2.

Population Size	Generations Run	Best Fitness	Generation Discovered
20	10000	1085287	6284
50	10000	2023.009	6
100	10000	2023.012	2266
250	10000	0.051	13
500	5944	306.003	11
1000	3337	0.046	10
5000	1000	0.047	18
10000	535	0.033	14

Table 6. Results of Series Expansion Runs with a Probability of Mutation of 0.001

In these initial runs, smaller starting populations (20 and 50), even though they reached the termination criteria of 10,000 generations, never achieved as high a fitness as the larger populations. They continued to be tested, although in later runs they were eliminated.

Population Size	Generations Run	Best Fitness	Generation Discovered
20	10000	143258	48
50	10000	89559.43	8016
100	10000	2023.012	6108
250	10000	2023.002	2
500	10000	305.983	761
1000	1000	305.981	13
5000	391	0.038	23
10000	200	0.034	6

Table 7. Results of Series Expansion Runs with a Probability of Mutation of 0.01

Larger population sizes (5,000 and 10,000 individuals) evolving for longer generations usually caused a *java.io.exception* that stopped the run before reaching the predefined termination condition of 10,000 generations. Data was continuously collected up to the time of the failures, which explains the differences in the number of generations each population size was run. Eventually, the 10,000 member population size was eliminated from testing because it did take significantly more time to complete, always crashed, and does not seem to provide significantly better results than a 5,000 member population.

Population Size	Generations Run	Best Fitness	Generation Discovered
20	10000	2023.012	1255
50	10000	50.985	2081
100	10000	2023.012	17
250	10000	306.003	6
500	10000	305.970	1792
1000	1000	0.030	13
5000	940	0.048	23

Table 8. Results of Series Expansion Runs with a Probability of Mutation of 0.1

Population Size	Generations Run	Best Fitness	Generation Discovered
100	10000	522.500	2556
250	10000	3.424	9292
500	2704	80.890	1862
1000	1000	0.044	52
5000	1000	0.055	12

Table 9. Results of Series Expansion Runs with a Probability of Mutation of 1.0

From looking at the tables and comparing the best fitness values found against when they were discovered, populations of 1,000 and 5,000 achieved better results in fewer generations. With the exception of Table 7, the initial population size of 1,000 always achieved slightly better fitness than that of 5,000 and did not crash within 1,000 generations. Thus, the remaining runs were done using population sizes of 1,000 evolved for 1,000 generations.

Two additional control runs were performed after deciding on a population and generation size. The first run had its probability of mutation set to zero in order to see just the effects of crossover. The second run was done to see the results of a 50% chance of a mutation. The results are presented in Table 10.

Probability of Mutation	Population Size	Generations Run	Best Fitness	Generation Discovered
0	1000	1000	305.990	369
0.5	1000	1000	2.944	724

Table 10. Series Expansion Runs with Probabilities of Mutation of 0.0 and 0.5

The first runs conducted used a 0.001 probability of mutation. Figure 4 shows the graph of the average population fitness. For readability, only the results to 1,000 generations are displayed. The figure shows a lot of variance in the average population fitness from generation to generation[22]. The number of outliers increased with increased population size. As a result, medians were used to measure central tendency. Figure 5 shows the results. Population sizes of 500 and 10,000 were removed from the graph because their median values ($1.4 \text{ X } 10^7$ and $7.02 \text{ X } 10^6$) caused the other values to be unreadable.

[22] The outlier in Figure 4 plotted between generation 144 and 180 has a value greater than $1 \text{ X } 10^{38}$.

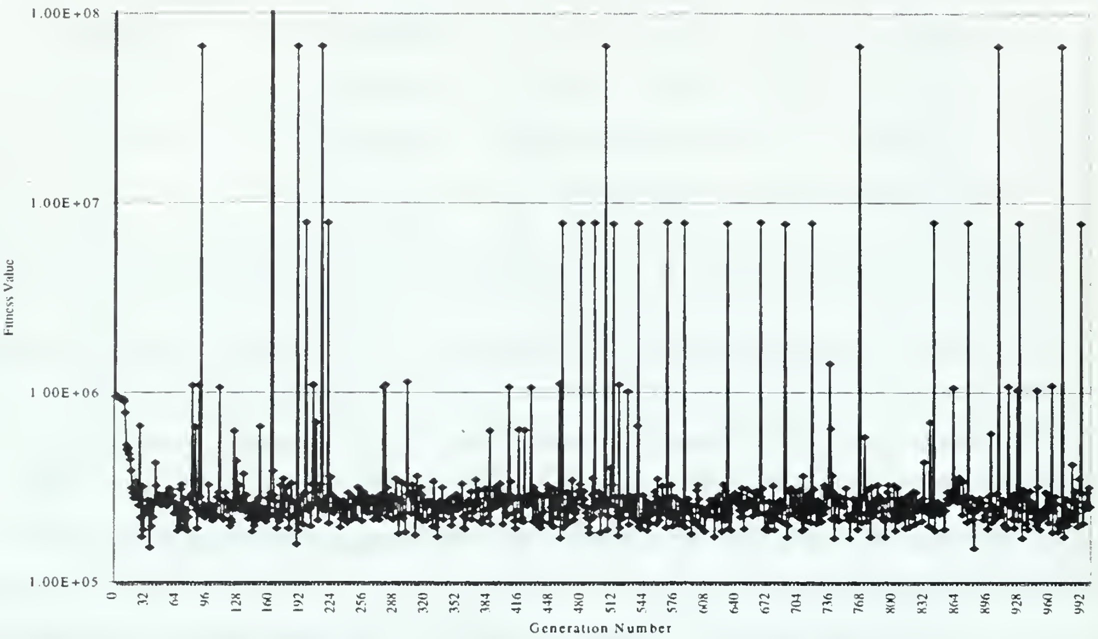


Figure 4. Average Population Fitness for One Run of a Population Size of 20 with a Probability of Mutation of 0.001

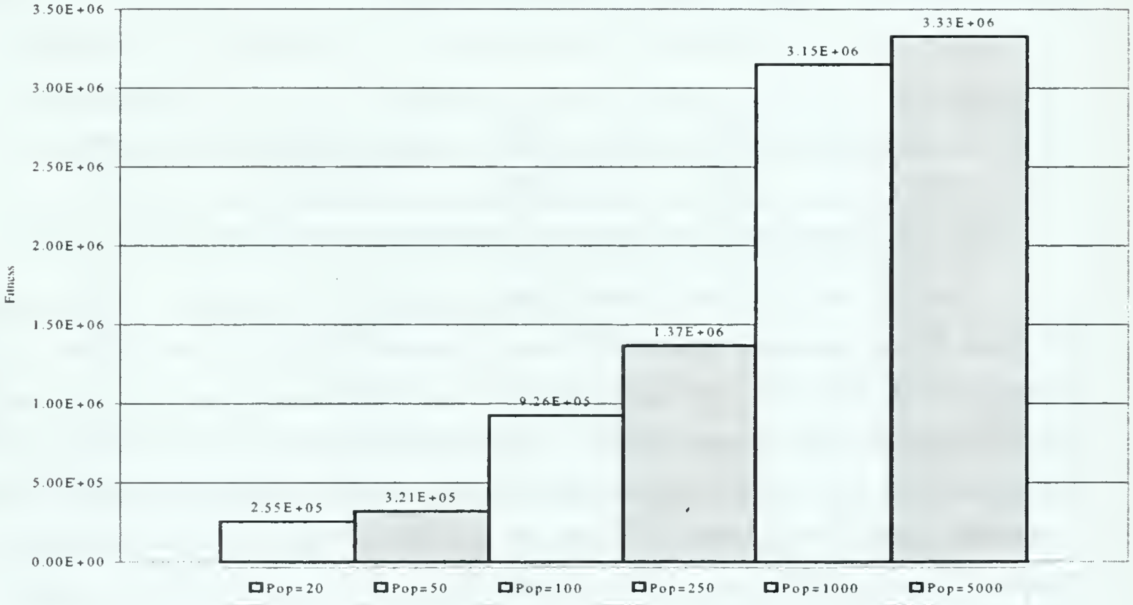


Figure 5. Average Fitness Values with a Probability of Mutation of 0.001

There is a minimum number of operations that must be performed in order to find a good solution to a series expansion, so the best-fit individual's complexity should be the closest to that minimum (see Chapter V). In Figure 6, one can see that the average population complexity is very close to the best-fit complexity, but always less. That trend is consistent across all the tests regardless of population size, probability of mutation, or number of generations. Table 11 shows the average population complexity compared to the best-fit individual's complexity. As the worst-fit individual's data was not collected in the control runs, one can only assume from this trend that individuals with significantly lower complexity are the least fit.

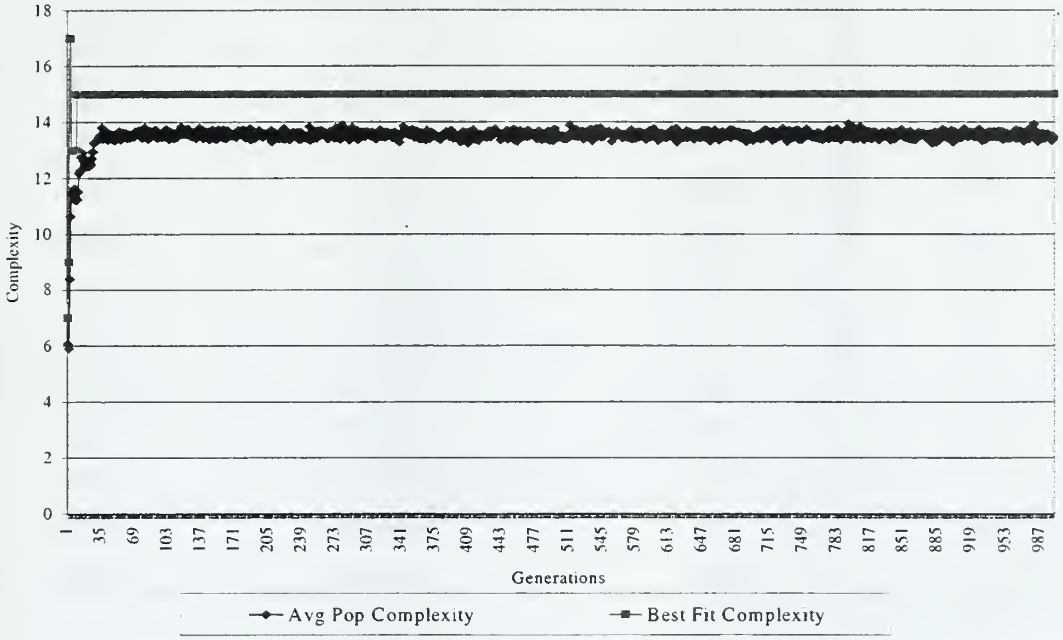


Figure 6. Comparison of Average Complexity with the Best-Fit Individual's Complexity for Population Size of 1,000 and a 0.001 Probability of Mutation

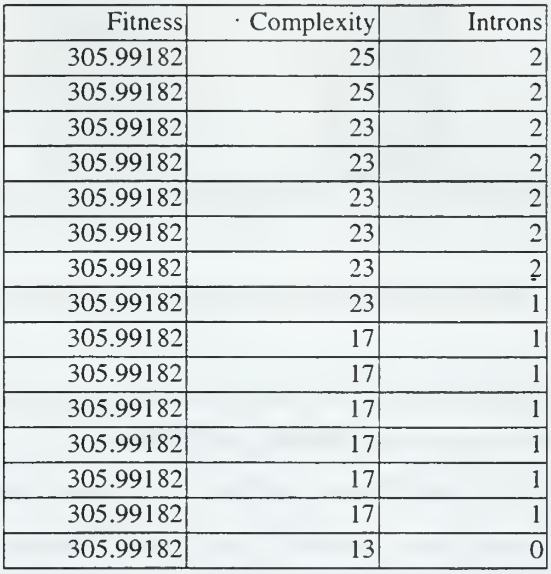
Probability of Mutation	Population Size	Average Population Complexity	Best Individual's Average Complexity
0.001	20	40.284	45.729
0.001	50	11.845	13.005
0.001	100	11.376	12.550
0.001	250	13.529	15.006
0.001	500	11.790	13.002
0.001	1000	13.529	14.993
0.001	5000	20.175	22.966
0.01	20	37.998	43.351
0.01	50	46.260	54.118
0.01	100	10.703	11.778
0.01	250	13.062	14.418
0.01	500	13.683	15.001
0.01	1000	13.566	14.994
0.01	5000	20.457	23.133
0.1	20	13.621	14.948
0.1	50	24.191	27.417
0.1	100	15.130	16.767
0.1	200	11.842	12.999
0.1	500	49.428	57.080
0.1	1000	25.666	29.665
0.1	5000	21.813	25.017
1.0	100	53.519	64.677
1.0	250	26.091	29.309
1.0	500	65.597	81.181
1.0	1000	25.232	24.916*
1.0	5000	20.928	21.050
0.5	1000	59.941	74.692
0.0	1000	11.768	13.016

* This is the only inconsistent point

Table 11. Comparison of Average Population Complexities to Best-Fit Complexities

B. GEOMETRIC SERIES EXPANSION WITH INTRONS

In order to answer the questions posed in Chapter IV, namely whether or not adding explicitly defined introns function correctly, the author needed to find examples of individuals in which the number of introns changed, but the fitness remained the same. Figure 7 is an excerpt from the Excel^{TM} spreadsheet into which all collected data was imported. It, along with other examples, show that adding explicitly defined introns within the GPSYS-1.1 does work. The next question to answer is what effect does it have when applied to the same problem as the control runs.



Fitness	Complexity	Introns
305.99182	25	2
305.99182	25	2
305.99182	23	2
305.99182	23	2
305.99182	23	2
305.99182	23	2
305.99182	23	2
305.99182	23	1
305.99182	17	1
305.99182	17	1
305.99182	17	1
305.99182	17	1
305.99182	17	1
305.99182	17	1
305.99182	13	0

Figure 7. Excerpt from Run Output Showing Explicitly Defined Introns

With a population size of 1,000 and 1,000 generations evolved, adding introns produced the best fitness values depicted in Figure 8. Comparing these values to the best fitness found in Tables 6 through 10 from the control runs, the fitness values depicted in the figure most closely match results from control runs with a population size of 500. Those control run results were not very good, which is why 500 was not chosen as the original population size for further tests.

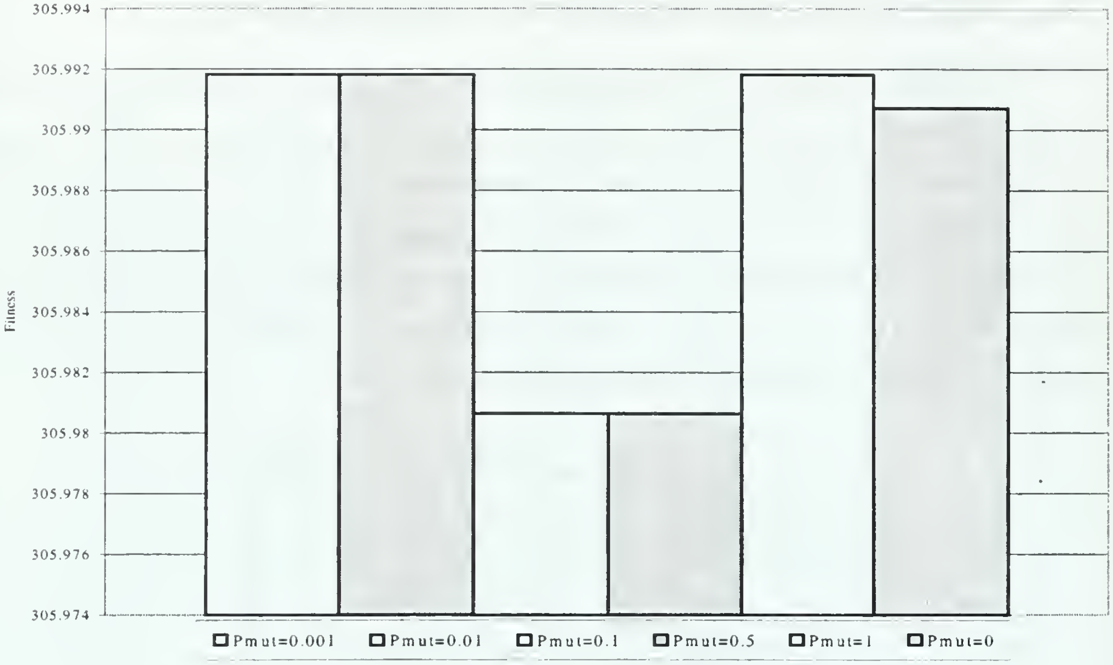


Figure 8. Best Individuals with Introns for Population Size of 1,000

Since these results were not as good as those from the controls, it was decided to run tests with a population size of 500 to see if there was any improvement in the best fitness found. The only population to produce a good answer was the one with a probability of mutation of 0.5. Because of this encouraging result, two additional tests were run with a population sizes of 250 and 750. These tests were done in an attempt to zero in on the best population size to use for the experiment, since 1,000 performed poorly. However, they produced best fitnesses of 305.990 and 305.985 respectively (see Figure 9).

With no consistent discovery of a highly fit individual as compared to the control runs, the obvious conclusion is that population sizes of 500 and 1,000 with explicitly defined introns run for 1,000 generations do not render as good a solution. The addition of introns appears to causes the population to converge on a local optima. As can be seen in Table 12, this convergence occurs very rapidly. However, the introns prevented the extreme variance in average population fitness similar to what was depicted in Figure 6. The results from one run are shown in Figure 10.

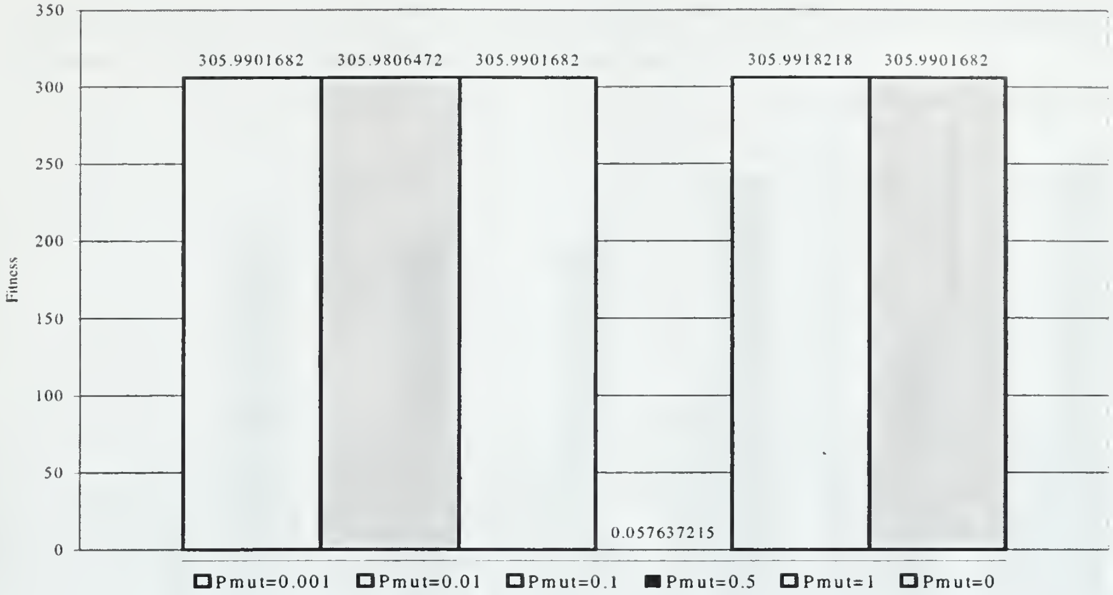


Figure 9. Best Individuals with Introns for Population Size of 500

Population Size	Probability of Mutation	Generation Discovered
1000	0.0	9
1000	0.001	6
1000	0.01	10
1000	0.1	6
1000	0.5	5
1000	1.0	3
500	0.0	3
500	0.001	3
500	0.01	3
500	0.1	6
500	0.5	8
500	1.0	4

Table 12. Generation of Discovery of Best Fitness

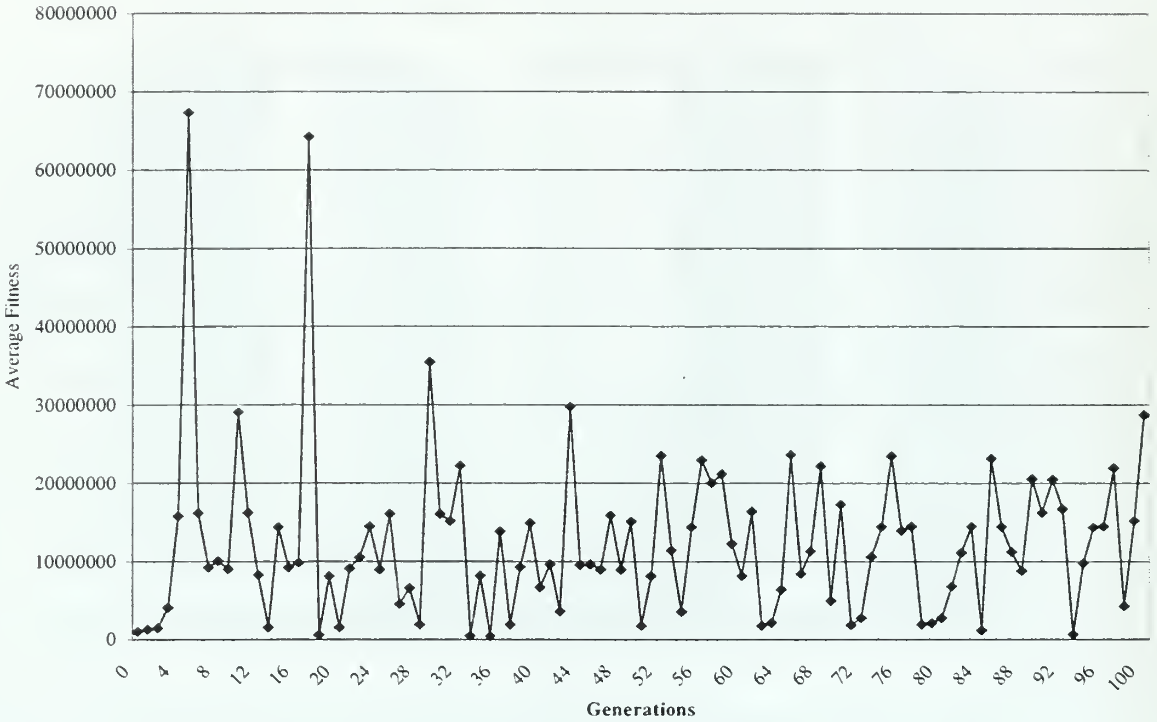


Figure 10. Single Run Average Population Fitness for Population Size 1,000 with a 0.001 Probability of Mutation

Table 11 compares average population complexity and the average best-fit individuals' complexity. The table shows that the best-fit individuals are almost without exception more complex than the population. Figure 11 is a comparison of the average population complexity with that of the best- and worst-fit individuals. One can see that in several bars the average population complexity remains smaller than the best-fit average complexity, but this not consistently the case. An important point though is that the best-fit complexity over all runs remains relatively the same. This similarity is probably due to the quick convergence to local optima in all the tests. Figure 8 shows all the local optima were very close in fitness value.

Since the fitness values of the best individuals converged so quickly to a local optima, were introns responsible? Figure 12 compares the average number of introns for a member of the population to that of both the average best and worst-fit individuals. In the runs with a probability of mutation of 0.001, 0.01, and 0.1, the best-fit individuals never had an intron. Additionally, very few introns were found within either the entire population or the worst individuals during those runs.

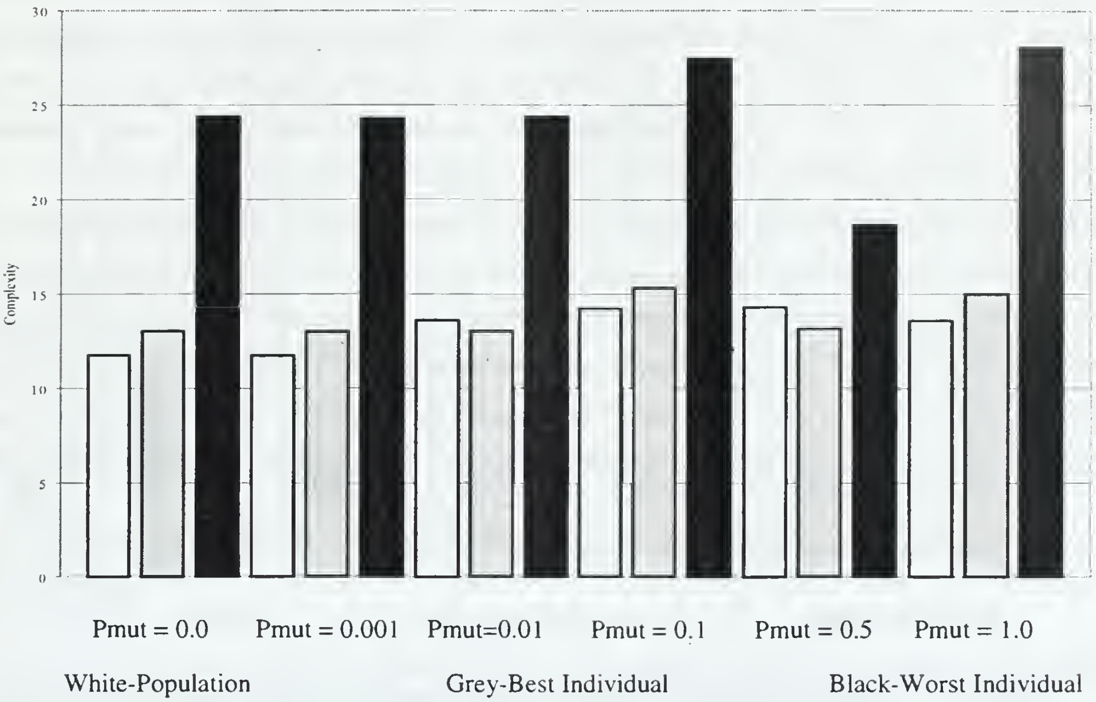


Figure 11. Average Complexities of the Population, Best-Fit, and Worst-Fit for Population Size of 1,000

Table 13 is a comparison of whether introns existed in the population, best individual, and worst individuals, and whether or not they were ultimately excised from the tree. Introns consistently resided within the worst-fit individual longer than the best-fit individual, but the worst-fit individual did not always have an intron. In most cases, the worst individual had no introns. In Figure 12, the worst-fit individual has fewer introns than the average population.

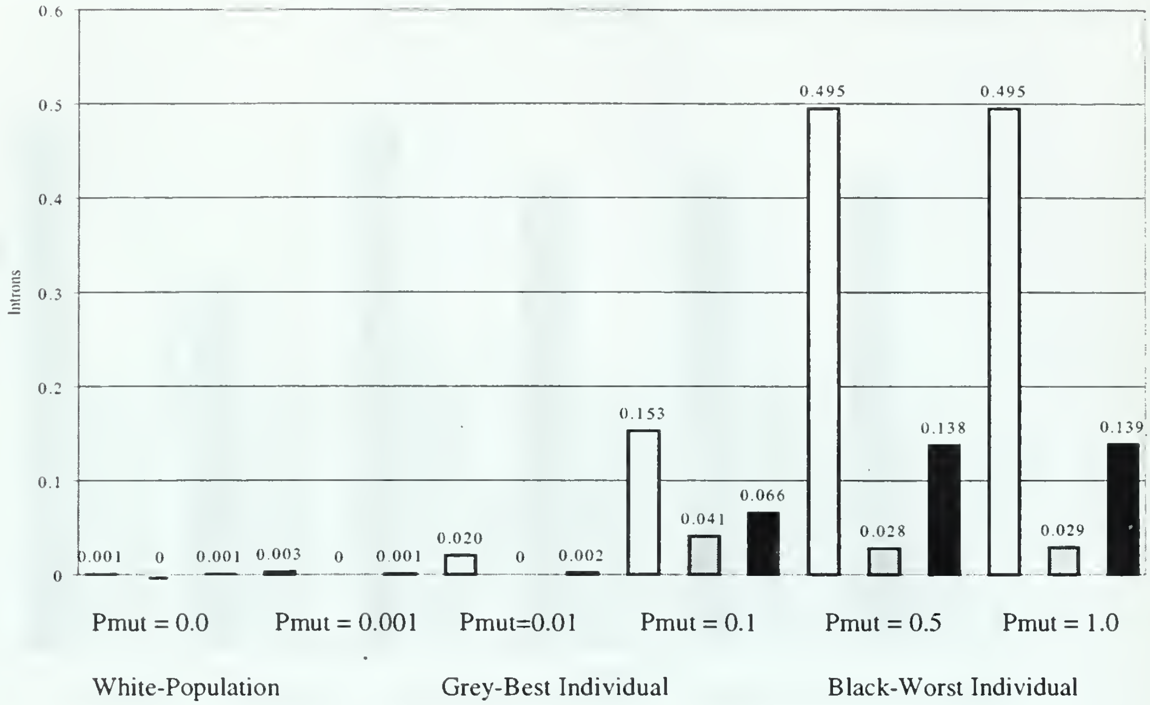


Figure 12. Average Number of Introns in the Population, Best Individual and Worst Individual

Probability of Mutation	Generation in which introns no longer found in population	Generation in which introns no longer found in best	Generation in which introns no longer found in worst
0	Introns throughout	20	127*
0.001	8	Never existed	1
0.01	Introns throughout	Never existed	1
0.1	Introns throughout	Never existed	1
0.5	Introns throughout	20	39
1	Introns throughout	20	84*

*Introns were found sporadically throughout the rest of the generations, but were primarily 0

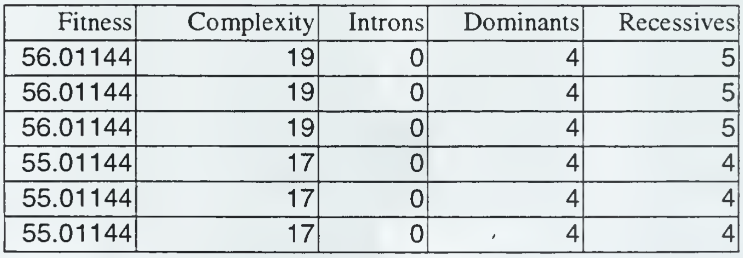
Table 13. Comparison of Existence of Introns

From the information provided in Chapter II, the chromosomes, or gene trees in this instance, are acting like the chromosomes of the lower eucaryotes which have lost most of their introns in favor of more continuous genes. The consistently low occurrence of introns throughout the population indicate that the genetic mechanisms, without specifically being directed to do so, replace the intron subtree. This function is consistent with the "splice point" role of introns in DNA recombination.

In the tests with high probabilities of mutation, the average population fitness show little variation as compared to runs with a low probability of mutation because the intron region is an area where these mutations occur without affecting the overall fitness of the individual[23]. Although there was one run that found a good solution, the general effect of adding introns is that of preventing the drastic changes resulting from crossover and mutation. With dramatic changes less likely, the population becomes more homogenous, which explains the quick and permanent convergence to local optima.

C. GEOMETRIC SERIES EXPANSION WITH DOMINANT/RECESSIVE GENES

In order to answer the questions posed in Chapter IV relating to whether the addition of dominant/recessive genes works, the author needed to find examples of individuals in which the number of recessive genes changed and the fitness increased. Figure 13 is an excerpt from the Excel™ spreadsheet into which all run data was imported. It, along with other examples, show that adding dominant/recessive genes within GPSYS-1.1 does work. Next, what effect do dominant/recessives have when applied to the same problem as were the control runs?



Fitness	Complexity	Introns	Dominants	Recessives
56.01144	19	0	4	5
56.01144	19	0	4	5
56.01144	19	0	4	5
55.01144	17	0	4	4
55.01144	17	0	4	4
55.01144	17	0	4	4

Figure 13. Excerpt from Run Output Showing Dominant and Recessive Genes

[23] Introns will affect fitness if complexity is considered during evaluation. If two individuals evolve to the same answer, the less complex individual will be given the higher fitness.

With a population size of 1,000 run for 200 generations, adding dominant and recessive genes produced the best fitness values as depicted in Figure 14. Only durations of 200 generations were run because early analysis of the output data on runs of 1,000 generations showed convergence on the same value at approximately 20 generations.

As compared to the runs with just the introns enabled (see Figure 8), dominant/recessive genes either had similar or better results (55.011 an 1.044). Similar tests were run for a population size of 500 with similar results (see Figure 15).

Table 12 notes that the addition of introns caused a quick convergence upon a local optima. Table 14 is a similar comparison but with a population including dominant and recessive genes. Convergence was not as quick as with the introns cases. Although more aligned with the fitness values from the control runs for population size of 1,000 (see Tables 6-10), the time for convergence for a population size of 500 is much faster than those reported from the control runs.

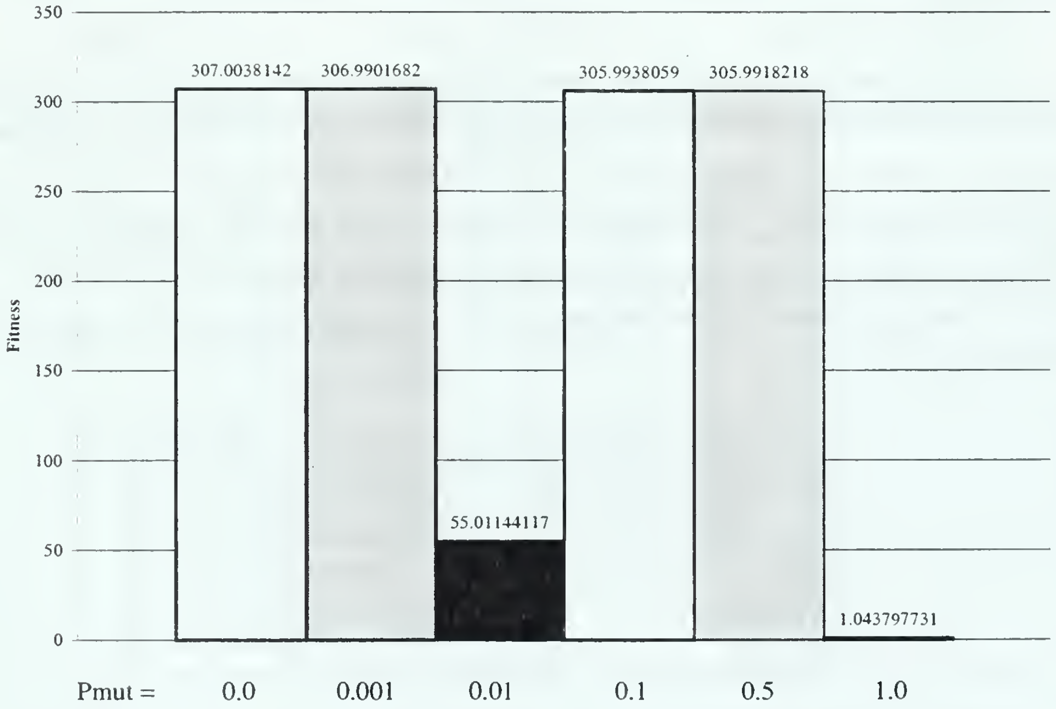


Figure 14. Best Individuals with Dominant/Recessive Genes in Population Size of 1,000

Population Size	Probability of Mutation	Generation Discovered
1000	0.0	3
1000	0.001	4
1000	0.01	22
1000	0.1	5
1000	0.5	25
1000	1.0	143
500	0.0	1
500	0.001	1
500	0.01	22
500	0.1	38
500	0.5	39
500	1.0	86

Table 14. Generation of Discovery of Best Fitness

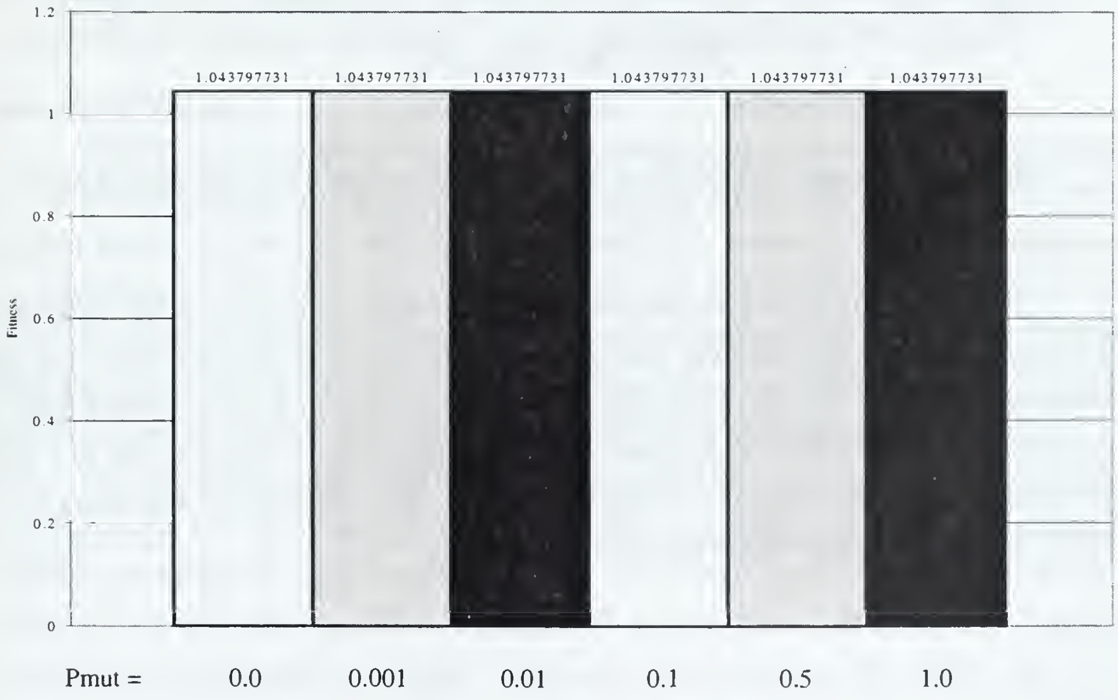


Figure 15. Best Individuals withDominant/Recessive Genes for Population Size of 500

Figures 16 and 17 compare the complexities of the average population, best-fit, and worst-fit individuals for population size of 1,000 and 500, respectively.

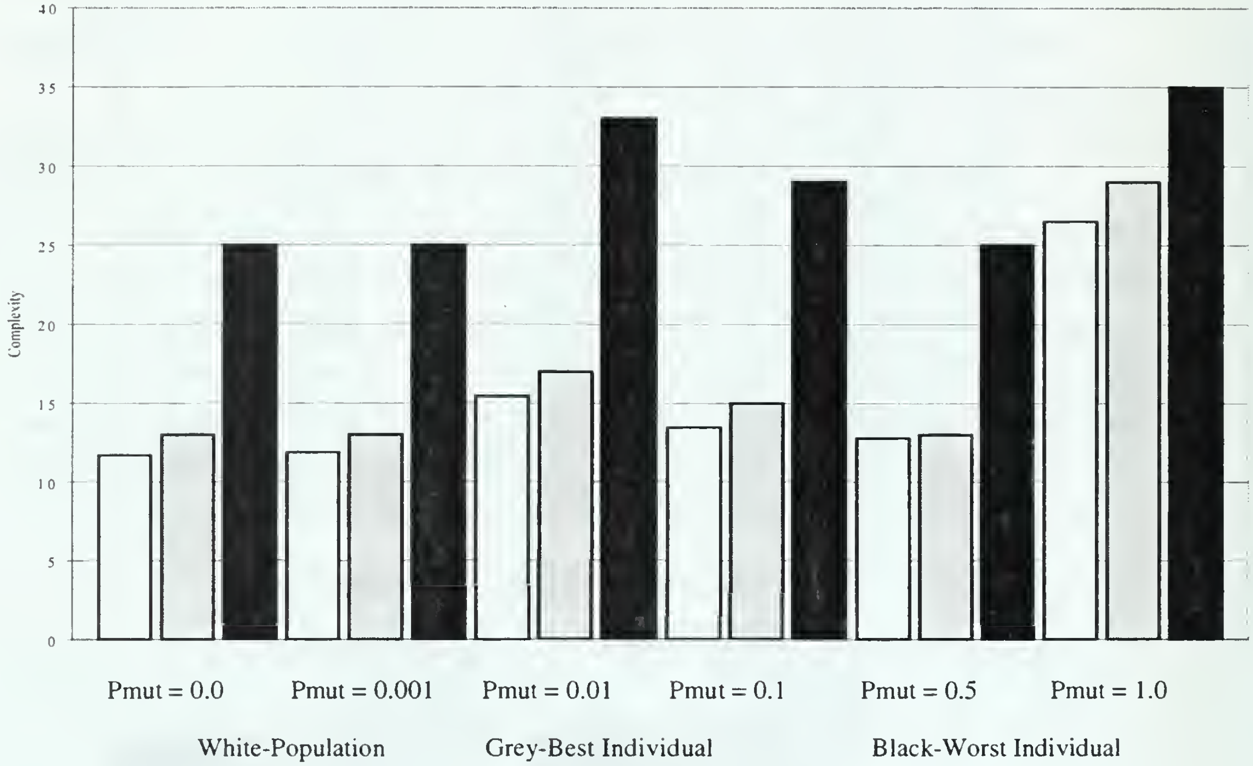


Figure 16. Average Complexities of the Population, Best-Fit, and Worst-Fit for Population Size of 1,000

The results in Figure 16 are consistent with the control runs in that the average population complexity is slightly less than the best fitness. Figure 17 shows the same trend, except for the run with a probability of mutation of 1. In Table 11, which listed the complexities of the best-fit individuals from the control runs, there was one run at that probability of mutation that also was not consistent. The difference in the values is very small, as was the case in the control runs.

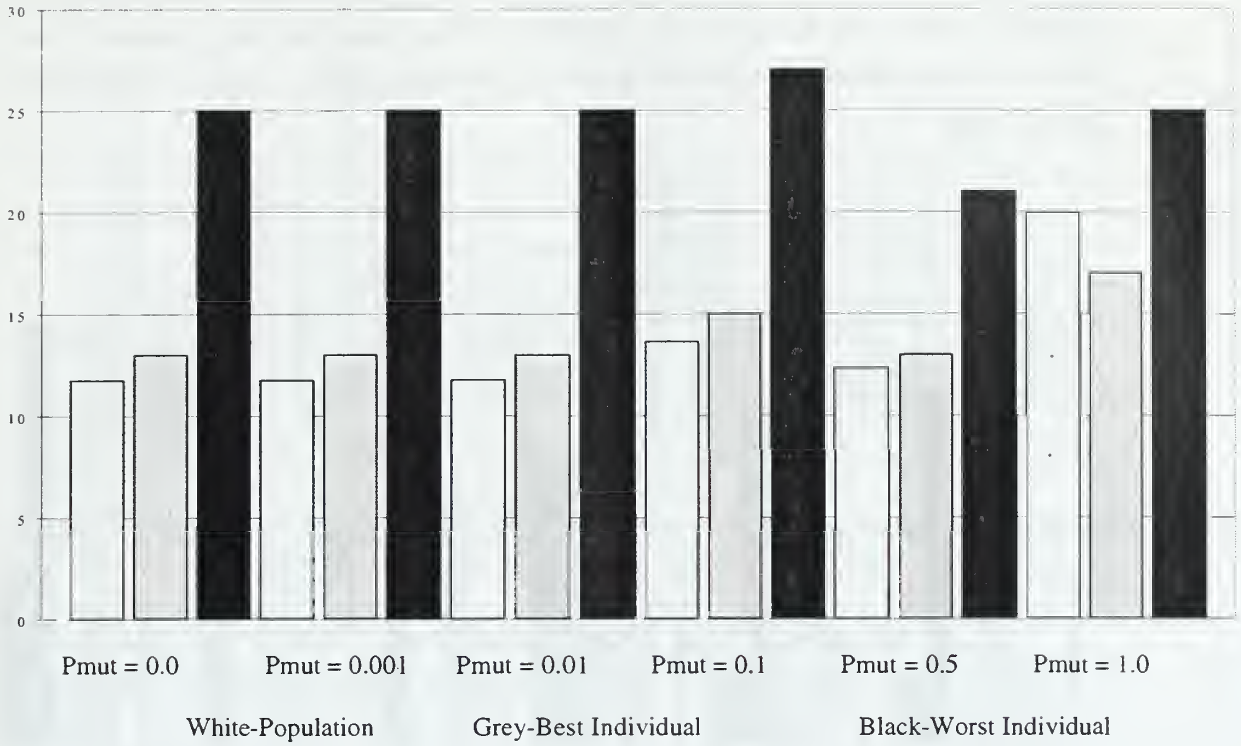


Figure 17. Average Complexities of the Population, Best-Fit, and Worst-Fit for Population Size of 500

Figure 18 is a comparison of the average number of dominant and recessive genes in the population, best-fit, and worst-fit individuals. The results are what one would expect; there are generally more recessive genes in the worst-fit individual, represented by black bars in the figure, than there are in the best and the average population. The very low recessive gene occurrence in the higher probabilities of mutation is puzzling. Both the crossover and mutation mechanisms have equal opportunities for exchanging a recessive gene for another recessive gene. So why is the occurrence so infrequent?

The dominant and recessive genes are modeled after Mendel's law of dominance (see Appendix C). As a result, crossover is similar to meoisis and sexual recombination and follows Mendel's law of segregation. In finite populations where Mendel's law of segregation holds, the population is subject to genetic drift, a dispersive evolutionary force that removes genetic variation. Mutation usually counters its effects, but in this problem many of the mutations are neutral because the end products are functionally the same because a recessive gene can replace a dominant one and vice versa. Thus, this particular implementation follows the neutral theory and is an example of Non-Darwinian evolution. Although the crossover probabilities were modeled after Mendel's law, the type of dominance being exhibited is that of underdominance, which is influenced by a genetic drift.

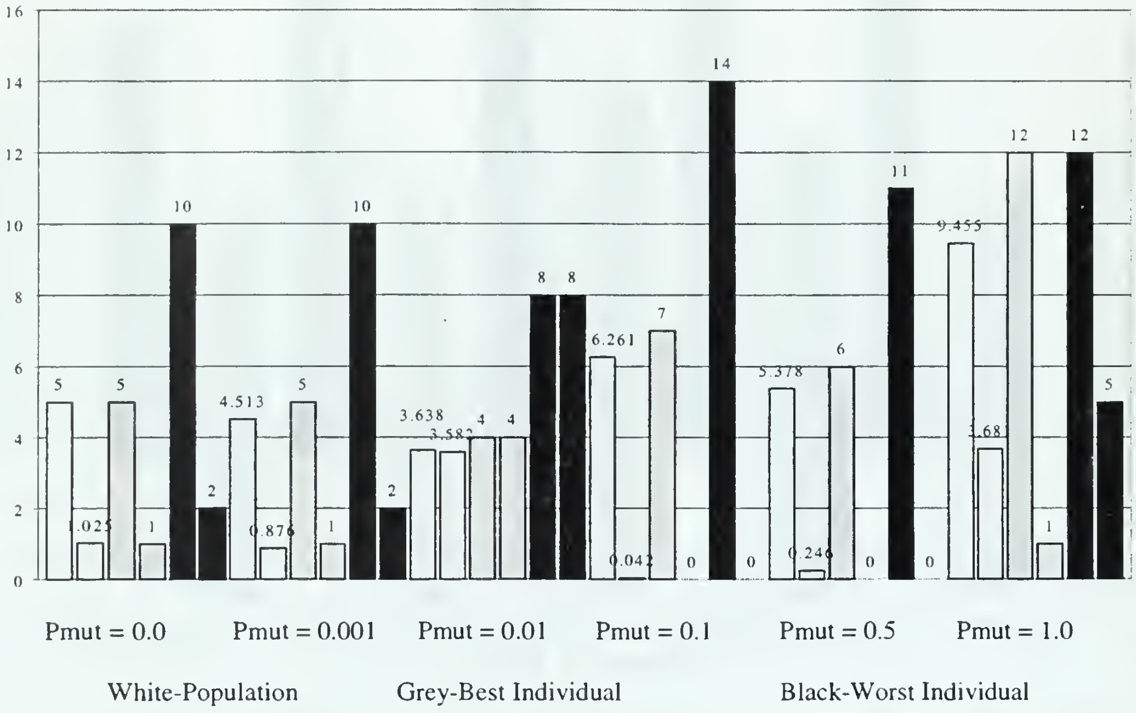


Figure 18. Average Number of Dominant and Recessive Genes in the Population, Best Individual and Worst Individual for Population Size of 1,000

One feature that populations which include dominant and recessive genes share with those which include introns is that they prevent the large variance in average population fitness seen in the control runs. Figure 19 is a typical example showing that the average population fitness falls within a smaller range.

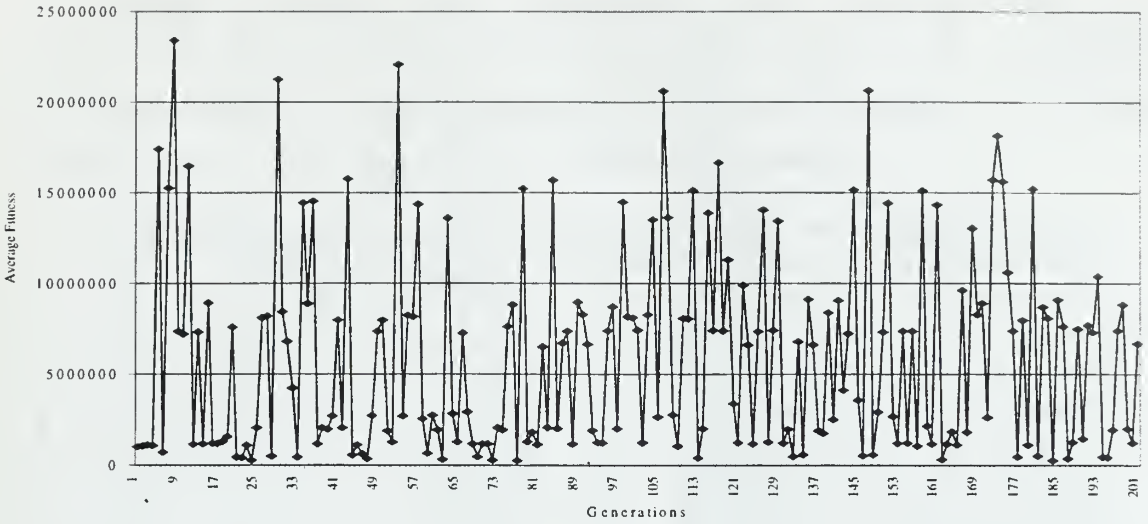


Figure 19. Single Run Average Population Fitness over 200 Generations for a Population Size of 1,000 with Probability of Mutation of 0.1

D. GEOMETRIC SERIES EXPANSION WITH BOTH INTRONS AND DOMINANT/RECESSIVE GENES

Although only based on runs using the same random seed, the results of the previous sections provide enough information about the effect of adding introns or dominant/recessive genes such that test results from the combination of the two can be better explained. This thesis is most interested in determining what happens when both mechanisms are used in the same population. As such, twenty-one random seeds were used to run the tests so that a statistically significant result could be found.

One overall finding is that the average population fitness did not exhibit the disparity manifest in the control runs. Medians continued to be used to measure central tendency for consistency with earlier experiments.

The following tables depict the best-fit individual found for population sizes of 500 and 1000 evolved for 200 generations using the six previously mentioned probabilities of mutation. Following these are the results for the entire population in which that best-fit individual was found.

Generation Found	Fitness	Complexity	Introns	Dominants	Recessives
19	1.051	15	0	6	1
27	2.053	23	0	9	2
50	3.038	23	0	8	3
27	3.051	15	0	4	3
11	305.999	13	0	6	0
199	305.999	13	0	6	0
8	306.003	13	0	6	0
16	306.007	13	0	6	0
7	306.990	13	0	5	1
10	306.990	13	0	5	1
14	306.992	13	0	5	1
4	307.007	13	0	5	1
15	307.992	15	0	5	2
4	307.993	15	0	5	2
10	308.985	15	0	4	3
24	308.985	15	0	4	3
8	308.992	15	0	4	3
3	309.001	15	0	4	3
50	309.013	17	0	5	3
4	309.993	15	0	3	4
3	2023.012	13	0	6	0
Median: 11	306.992	15	0	5	1

Table 15. Best-Fit Individuals for Population Size of 500 with Probability of Mutation of 0.0

The distinct absence of introns in all the best-fit individual was at first a cause for possible concern because the tests run with only introns had at least one intron in the best individual (see Table 13). Out of these runs, only 3 ever had a best-fit individual with an intron at any time and that intron was removed early in the run. More noteworthy is that introns were completely removed from the entire population for every random seed tested (see Tables 17 and 18).

The excision of the some introns is consistent with the findings reported in a section 2, but the complete removal of introns in populations with a low probability of mutation shows the effect of genetic drift caused by the dominant and recessive genes. Genetic drift is also responsible for the removal of all recessive genes as well. Additionally, the tournament selection process would also rid the population of a portion of individuals with introns and recessives because the least fit individual selected in any tournament is the one that will be replaced as a result of crossover.

Tables 19-22, which show the results from using a 0.001 probability of mutation. Tables 23-26, which show the results from using a 0.01 probability of mutation, show nearly identical results. The small probability of mutation is not strong enough to counter the effects of the drift. The tables are provided for completeness.

Generation Found	Fitness	Complexity	Introns	Dominants	Recessives
7	306.001	13	0	6	0
22	306.006	13	0	6	0
9	306.007	13	0	6	0
26	305.994	15	0	7	0
9	305.999	15	0	7	0
8	305.994	17	0	8	0
4	306.999	13	0	5	1
5	306.999	13	0	5	1
153	306.999	13	0	5	1
17	307.001	13	0	5	1
26	307.001	13	0	5	1
11	307.003	13	0	5	1
7	307.004	13	0	5	1
19	307.007	13	0	5	1
21	306.993	15	0	6	1
2	307.001	15	0	6	1
9	307.003	15	0	6	1
28	2.039	17	0	6	2
9	308.985	15	0	4	3
31	3.051	17	0	5	3
3	56.013	19	0	4	5
Median: 9	306.999	13	0	5	1

Table 16. Best-Fit Individual for Population Size of 1,000 with Probability of Mutation of 0.0

Median Population Fitness	Median Population Complexity	Median Population Dominants	Median Population Recessives	Generation in which Recessives Disappeared from Population	Median Population Introns	Generation in which Introns Disappeared from Population
5139698	11.788	5.384	0	21	0	2
3264676	11.828	5.404	0	19	0	2
13750212	11.784	4.472	0.908	Never	0	2
1936911	13.496	5.334	0.912	Never	0	2
14103352	13.520	4.496	1.77	Never	0	2
2248451	13.488	3.570	2.68	Never	0	2
3715621	13.532	3.584	2.688	Never	0	2
17377561	13.616	3.602	2.702	Never	0	2
156755	20.692	7.112	2.706	Never	0	2
12918800	11.744	5.358	0	29	0	3
2219762	11.816	4.522	0.886	Never	0	3
12954467	11.756	4.448	0.912	Never	0	3
5165956	11.764	4.448	0.928	Never	0	3
11949839	11.768	4.454	0.93	Never	0	3
636449	20.284	7.840	1.804	Never	0	3
17612654	13.624	3.636	2.672	Never	0	3
3479845	15.220	4.408	2.688	Never	0	3
156755	20.692	7.112	2.706	Never	0	4
13114036	11.752	3.582	1.782	Never	0	10
15879139	11.784	2.698	2.694	Never	0	14
5263248	11.772	4.454	0.914	Never	0	41
Median: 5165956	11.828	4.454	1.77		0	3

Table 17. Population Statistics for Population Size of 500 with Probability of Mutation of 0.0

Median Population Fitness	Median Population Complexity	Median Population Dominants	Median Population Recessives	Generation in which Recessives Disappeared from Population	Median Population Introns	Generation in which Introns Disappeared from Population
8908048	11.796	5.378	0	36	0	2
4727214	11.822	5.405	0	23	0	2
8908555	13.512	6.253	0	28	0	2
15611664	11.792	4.491	0.892	Never	0	2
15361321	11.776	5.387	0	24	0	3
13383132	11.786	4.502	0.884	Never	0	3
15853824	11.776	4.497	0.888	Never	0	3
4471257	11.810	4.503	0.898	Never	0	3
9729269	11.786	4.485	0.907	Never	0	3
3985730	13.516	3.572	2.678	Never	0	3
15584882	15.106	5.293	1.764	Never	0	4
14972892	16.966	3.544	4.439	Never	0	4
9885344	11.776	5.378	0	22	0	5
14791074	11.764	4.472	0.907	Never	0	5
9006572	13.562	5.383	0.895	Never	0	6
9605791	11.784	4.475	0.917	Never	0	6
23048590	13.630	3.560	2.756	Never	0	7
15879818	11.748	4.455	0.907	Never	0	10
9052629	13.550	6.268	0	44	0	11
10484363	11.776	3.554	1.827	Never	0	15
9842645	11.786	4.485	0.904	Never	0	20
Median: 9885344	11.792	4.497	0.898		0	4

Table 18. Population Statistics for Population Size of 1,000 with Probability of Mutation of 0.0

Generation Found	Fitness	Complexity	Introns	Dominants	Recessives
11	306.006	13	0	6	0
11	306.007	13	0	6	0
11	2023.012	13	0	6	0
178	305.981	15	0	7	0
14	306.004	15	0	7	0
7	306.990	13	0	5	1
103	306.990	13	0	5	1
29	306.992	13	0	5	1
16	307.007	13	0	5	1
2	1.051	15	0	6	1
11	306.981	15	0	6	1
6	307.992	13	0	4	2
5	308.003	13	0	4	2
21	308.003	13	0	4	2
4	308.009	13	0	4	2
38	2.051	15	0	5	2
13	307.992	15	0	5	2
42	2.039	17	0	6	2
3	309.001	15	0	4	3
28	3.038	23	0	8	3
34	3.038	23	0	8	3
Median: 13	306.990	13	0	5	1

Table 19. Best-Fit Individual for Population of 500 with Probability of Mutation of 0.001

Generation Found	Fitness	Complexity	Introns	Dominants	Recessives
26	305.990	13	0	6	0
33	305.990	13	0	6	0
20	306.007	13	0	6	0
19	305.990	15	0	7	0
13	305.994	15	0	7	0
6	306.990	13	0	5	1
15	306.992	13	0	5	1
4	307.011	13	0	5	1
8	2024.004	13	0	5	1
8	2024.012	13	0	5	1
8	306.993	15	0	6	1
18	306.994	15	0	6	1
16	306.999	15	0	6	1
4	307.990	13	0	4	2
3	308.004	13	0	4	2
2	308.032	15	0	5	2
25	2.039	17	0	6	2
27	53.022	19	0	7	2
16	3.046	15	0	4	3
24	3.051	15	0	4	3
31	3.051	17	0	5	3
Median: 16	306.990	15	0	5	1

Table 20. Best-Fit Individual for Population Size of 1,000 with Probability of Mutation of 0.001

Median Population Fitness	Median Population Complexity	Median Population Dominants	Median Population Recessives	Generation in which Recessives Disappeared from Population	Median Population Introns	Generation in which Introns Disappeared from Population
13094926	11.756	3.584	1.786	Never	0	10[*]
14493739	11.784	2.698	2.692	Never	0	15[*]
15554173	11.768	5.372	0	31[*]	0	2[*]
4560987	11.780	5.380	0	22[*]	0	2[*]
3108047	11.824	5.404	0	22[*]	0	2[*]
14484776	13.580	5.408	0.888	Never	0	2[*]
14276082	11.780	4.474	0.908	Never	0	2[*]
2489404	11.804	4.480	0.910	Never	0	2[*]
14321856	11.760	4.202	1.156	Never	0	2[*]
17111365	11.712	3.582	1.766	Never	0	2[*]
15568897	15.160	5.288	1.766	Never	0	2[*]
14365302	11.760	3.590	1.792	Never	0	2[*]
2210528	13.472	4.444	1.798	Never	0	2[*]
13988350	11.700	3.584	1.776	Never	0	22[*]
2004814	13.504	5.364	0.892	Never	0	23[*]
4786412	11.752	5.370	0	27[*]	0	3[*]
6691426	11.724	4.460	0.880	Never	0	3[*]
3541954	13.624	5.376	0.946	Never	0	3[*]
12942572	11.784	3.582	1.808	Never	0	3[*]
152967	20.688	7.152	2.696	Never	0	3[*]
312936	20.492	7.034	2.700	Never	0	4[*]
Median: 12942572	11.784	4.480	1.156		0	2

[*] Introns or recessive genes were found sporadically throughout the rest of the generations, but were primarily absent.

Table 21. Population Statistics for Population Size of 500 with Probability of Mutation of 0.001

Median Population Fitness	Median Population Complexity	Median Population Dominants	Median Population Recessives	Generation in which Recessives Disappeared from Population	Median Population Introns	Generation in which Introns Disappeared from Population
8828642	11.748	4.470	0.902	Never	0	10[*]
9760143	11.794	5.394	0	36[*]	0	11[*]
16008376	15.096	5.303	1.742	Never	0	11[*]
9809607	11.776	3.603	1.783	Never	0	12[*]
15432350	11.730	3.587	1.776	Never	0	13[*]
25276858	11.806	3.577	1.828	Never	0	13[*]
8656193	13.540	6.267	0	32[*]	0	2[*]
8865615	13.544	5.390	0.883	Never	0	2[*]
8411607	13.552	5.377	0.897	Never	0	2[*]
23651367	11.784	4.498	0.893	Never	0	3[*]
23936360	11.854	4.483	0.947	Never	0	3[*]
4733644	13.526	3.566	2.684	Never	0	3[*]
3223203	13.482	3.565	2.685	Never	0	3[*]
10384052	11.774	5.373	0	46[*]	0	4[*]
3907847	11.818	5.395	0	32[*]	0	4[*]
3804003	13.568	3.584	2.690	Never	0	5[*]
9818730	11.750	4.450	0.910	Never	0	6[*]
10185169	11.778	4.481	0.911	Never	0	6[*]
16454039	16.896	6.180	1.773	Never	0	7[*]
10341839	11.748	4.443	0.929	Never	0	8[*]
3412313	13.464	6.231	0	35[*]	0	9[*]
Median: 9809607	11.818	4.483	0.911		0	6

[*] Introns or recessive genes were found sporadically throughout the rest of the generations, but were primarily absent.

Table 22. Population Statistics for Population Size of 1,000 with Probability of Mutation of 0.001

Generation Found	Fitness	Complexity	Introns	Dominants	Recessives
23	305.990	13	0	6	0
22	305.992	13	0	6	0
8	306.003	13	0	6	0
16	2023.012	13	0	6	0
19	0.051	15	0	7	0
13	305.981	15	0	7	0
5	305.999	15	0	7	0
3	306.992	13	0	5	1
143	307.003	13	0	5	1
14	306.992	15	0	6	1
30	306.993	15	0	6	1
7	306.994	15	0	6	1
18	306.994	15	0	6	1
10	2024.012	15	0	6	1
3	2025.009	13	0	4	2
59	2.051	15	0	5	2
70	308.003	15	0	5	2
8	308.006	15	0	5	2
7	2.055	17	0	6	2
88	308.990	15	0	4	3
7	309.985	15	0	3	4
Median: 14	306.992	15	0	6	1

Table 23. Best-Fit Individual for Population of 500 with Probability of Mutation of 0.01

Generation Found	Fitness	Complexity	Introns	Dominants	Recessives
9	305.990	13	0	6	0
21	305.992	13	0	6	0
169	305.992	13	0	6	0
10	305.999	13	0	6	0
21	305.999	13	0	6	0
16	306.001	13	0	6	0
5	306.003	13	0	6	0
12	306.003	13	0	6	0
51	0.051	15	0	7	0
17	305.992	15	0	7	0
15	305.994	15	0	7	0
5	306.992	13	0	5	1
13	306.999	13	0	5	1
59	307.003	13	0	5	1
33	307.007	13	0	5	1
7	306.981	15	0	6	1
14	306.981	15	0	6	1
140	306.990	15	0	6	1
16	1.053	23	0	10	1
6	2.051	15	0	5	2
20	2.051	15	0	5	2
Median: 16	305.999	13	0	6	0

Table 24. Best-Fit Individual for Population Size of 1,000 with Probability of Mutation of 0.01

Median Population Fitness	Median Population Complexity	Median Population Dominants	Median Population Recessives	Generation in which Recessives Disappeared from Population	Median Population Introns	Generation in which Introns Disappeared from Population
4459363	11.748	5.364	0.004	Never*	0	Never*
5036917	11.780	5.368	0.004	Never*	0	Never*
13270204	11.800	5.384	0.004	Never*	0	Never*
14458805	13.616	6.296	0.004	Never*	0	Never*
13003576	11.784	5.360	0.006	Never*	0	Never*
6371712	11.784	5.368	0.006	Never*	0	Never*
2921386	13.496	6.236	0.006	Never*	0	Never*
14222781	13.540	5.368	0.892	Never	0	Never*
3125493	11.840	4.518	0.894	Never	0	Never*
3929052	13.592	5.384	0.894	Never	0	Never*
14107135	11.776	4.482	0.902	Never	0	Never*
12941270	13.576	5.372	0.910	Never	0	Never*
5112462	11.752	4.450	0.914	Never	0	Never*
7816937	11.848	4.458	0.954	Never	0.012	Never
5044470	11.812	3.626	1.780	Never	0	Never*
16886785	13.460	4.438	1.790	Never	0	Never*
2095219	13.588	4.492	1.796	Never	0	Never*
2267586	13.568	4.460	1.814	Never	0	Never*
15452469	11.820	3.574	1.832	Never	0	Never*
6636942	11.832	2.648	2.864	Never	0	Never*
17868211	13.628	2.756	3.560	Never	0	Never*
Median: 6636942	11.840	4.518	0.902		0	

*Introns or recessive genes were found sporadically throughout the rest of the generations, but were primarily absent.

Table 25. Population Statistics for Population Size of 500 with Probability of Mutation of 0.01

Median Population Fitness	Median Population Complexity	Median Population Dominants	Median Population Recessives	Generation in which Recessives Disappeared from Population	Median Population Introns	Generation in which Introns Disappeared from Population
15254297	11.786	5.373	0.004	Never	0.001	Never
9278676	11.778	5.377	0.004	Never	0.001	Never
9092490	11.792	5.381	0.004	Never	0.001	Never
10378340	11.800	5.384	0.004	Never	0.001	Never
9932836	11.738	5.350	0.005	Never	0.001	Never
11085528	11.756	5.363	0.005	Never	0.001	Never
9416659	11.766	5.365	0.005	Never	0.001	Never
10419268	11.774	5.372	0.005	Never	0.001	Never
8481694	13.548	6.262	0.005	Never	0.001	Never
3133824	13.490	6.214	0.007	Never	0.001	Never
8781014	11.786	4.495	0.889	Never	0.001	Never
8513889	13.580	5.401	0.891	Never	0.001	Never
10222374	11.736	4.469	0.893	Never	0.001	Never
9542253	11.782	4.486	0.893	Never	0.001	Never
10376794	11.790	4.490	0.893	Never	0.001	Never
9115347	13.600	5.400	0.900	Never	0.001	Never
4446680	11.820	4.483	0.916	Never	0.001	Never
6356949	20.480	8.803	0.919	Never	0.001	Never
3255375	13.482	4.452	1.798	Never	0.001	Never
2590846	13.512	4.456	1.803	Never	0.002	Never
9745228	11.804	3.566	1.829	Never	0.001	Never
Median: 9278676	11.736	5.365	0.889		0.001	

Table 26. Population Statistics for Population Size of 1,000 with Probability of Mutation of 0.01

Tables 27 and 28 show the best individual found in tests with population sizes of 500 and 1,000, respectively. Tables 29 and 30 show the population statistics. Although Table 28 has one excellent solution, and several very good ones, the overall performance is poor compared to the results of the control runs. In fact, the values achieved are no better than the ones in previous tables. The reason for the disappointing results is the quick convergence caused by the introns. Although there are examples of the best-fit individual being found after the first 100 generations, the median generation of convergence is 10 to 16, respectively. The introns are prevalent enough to deter the beneficial effects of crossover that push the population towards better solutions.

Generation Found	Fitness	Complexity	Introns	Dominants	Recessives
8	305.992	13	0	6	0
3	306.003	13	0	6	0
19	306.003	13	0	6	0
128	305.985	15	0	7	0
8	305.994	15	0	7	0
19	305.994	15	0	7	0
70	306.003	15	0	7	0
3	306.990	13	0	5	1
41	306.990	13	0	5	1
3	306.992	13	0	5	1
39	306.992	13	0	5	1
3	307.003	13	0	5	1
9	2024.009	13	0	5	1
195	306.985	15	0	6	1
9	306.994	15	0	6	1
6	307.003	15	0	6	1
45	307.003	15	0	6	1
10	306.990	17	0	7	1
139	308.003	13	0	4	2
47	2.051	17	0	6	2
7	309.018	15	0	4	3
Median: 10	306.990	15	0	6	1

Table 27. Best-Fit Individual for Population Size of 500 with Probability of Mutation of 0.1

Generation Found	Fitness	Complexity	Introns	Dominants	Recessives
10	305.992	13	0	6	0
19	305.992	13	0	6	0
48	305.992	13	0	6	0
8	305.999	13	0	6	0
20	306.001	13	0	6	0
44	305.981	15	0	7	0
14	305.990	15	0	7	0
20	305.992	15	0	7	0
10	305.994	15	0	7	0
11	305.994	15	0	7	0
7	306.003	17	. 0	8	0
60	0.044	27	0	13	0
17	307.003	13	0	5	1
22	1.051	15	0	6	1
34	1.051	15	0	6	1
47	1.051	15	0	6	1
6	306.981	15	0	6	1
11	306.981	15	0	6	1
16	306.994	15	0	6	1
11	308.003	13	0	4	2
15	307.994	15	0	5	2
Median: 16	305.994	15	0	6	0

Table 28. Best-Fit Individual for Population Size of 1,000 with Probability of Mutation of 0.1

Median Population Fitness	Median Population Complexity	Median Population Dominants	Median Population Recessives	Generation in which Recessives Disappeared from Population	Median Population Introns	Generation in which Introns Disappeared from Population
17543695	13.684	5.444	0.904	Never	0.010	Never
14292803	11.860	4.486	0.944	Never	0.010	Never
16406734	13.640	4.504	1.822	Never	0.010	Never
13945383	13.688	3.616	2.728	Never	0.010	Never
4970352	11.816	5.346	0.042	Never	0.012	Never
14323678	11.876	5.390	0.042	Never	0.012	Never
14149692	13.628	6.248	0.044	Never	0.012	Never
13897494	11.896	5.378	0.046	Never	0.012	Never
3777730	13.620	6.264	0.048	Never	0.012	Never
3451521	11.888	5.324	0.062	Never	0.012	Never
13455803	11.888	4.506	0.926	Never	0.012	Never
12987416	11.912	4.518	0.928	Never	0.012	Never
4502141	11.888	4.482	0.936	Never	0.012	Never
3499190	13.632	5.360	0.948	Never	0.012	Never
3483406	11.876	4.480	0.950	Never	0.012	Never
4967723	11.808	4.434	0.954	Never	0.012	Never
7816937	11.848	4.458	0.954	Never	0.012	Never
13110608	11.880	4.464	0.954	Never	0.012	Never
4960141	11.868	4.440	0.962	Never	0.012	Never
2087360	11.912	3.602	1.848	Never	0.012	Never
2098868	13.688	4.456	1.852	Never	0.016	Never
Median: 7816937	11.888	4.504	0.944		0.012	

Table 29. Population Statistics for Population Size of 500 with Probability of Mutation of 0.1

Median Population Fitness	Median Population Complexity	Median Population Dominants	Median Population Recessives	Generation in which Recessives Disappeared from Population	Median Population Introns	Generation in which Introns Disappeared from Population
13877924	11.842	5.367	0.042	Never	0.011	Never
8321885	13.620	5.388	0.924	Never	0.011	Never
8359798	13.598	6.255	0.044	Never	0.012	Never
8589628	13.638	6.258	0.044	Never	0.012	Never
9616162	11.844	5.366	0.045	Never	0.012	Never
8474073	11.872	5.383	0.045	Never	0.012	Never
9640096	11.836	5.346	0.047	Never	0.012	Never
8150807	13.668	6.250	0.048	Never	0.012	Never
8403883	13.646	5.402	0.926	Never	0.012	Never
8528211	13.612	5.352	0.947	Never	0.012	Never
8955009	11.866	4.475	0.950	Never	0.012	Never
8267034	13.624	4.467	1.839	Never	0.012	Never
8953174	11.866	5.373	0.046	Never	0.013	Never
8974188	11.864	5.376	0.046	Never	0.013	Never
8881179	11.840	5.362	0.048	Never	0.013	Never
9084279	11.836	5.330	0.053	Never	0.013	Never
8647046	11.862	3.593	1.831	Never	0.013	Never
2475556	13.616	5.359	0.932	Never	0.014	Never
2714601	13.620	5.363	0.932	Never	0.015	Never
2609997	13.608	5.335	0.939	Never	0.015	Never
901914	20.466	9.635	0.060	Never	0.016	Never
Median: 8528211	13.598	5.366	0.053		0.012	

Table 30. Population Statistics for Population Size of 1,000 with Probability of Mutation of 0.1

The results in the remaining tables support the premise presented in Chapter II that introns counteract some of the effects of potentially harmful mutations. These runs had the highest probabilities of mutation and showed the best results.

Tables 31 and 35 show the best-fit individuals for a population size of 500 with probability of mutation of 0.5 and 1.0, respectively. While the median best-fit is similar to the results from previous runs, 12 out of the 42 runs found a good solution. A population size of 1,000, shown in Tables 32 and 36, achieves a far better median solution.

Generation Found	Fitness	Complexity	Introns	Dominants	Recessives
156	305.993	13	0	6	0
31	0.051	15	0	7	0
45	305.990	15	0	7	0
35	305.992	15	0	7	0
30	305.994	15	0	7	0
8	305.999	15	0	7	0
14	306.003	15	0	7	0
96	306.004	15	0	7	0
100	306.011	15	0	7	0
61	0.051	17	0	8	0
88	306.992	13	0	5	1
14	307.001	13	0	5	1
19	307.003	13	0	5	1
89	1.051	15	0	6	1
52	306.994	15	0	6	1
47	1.051	17	0	7	1
22	307.003	17	0	7	1
56	307.018	17	0	7	1
6	307.994	15	0	5	2
12	2.033	21	0	8	2
166	5.058	25	0	7	5
Median: 45	305.999	15	0	7	1

Table 31. Best-Fit Individual for Population Size of 500 with Probability of Mutation 0.5

Keeping track of the median complexities throughout all of the previous tables, the average best-fit complexity is 15 with a average population complexity of 11.8 and with little variance. It is possible to achieve a good result with a complexity of 15, but the examples of such are few. Looking back through the tables, one would see that the sporadic good results are often accompanied by a higher complexity. In Tables 31, 32, 35 and 36, the runs that achieved a good result have higher complexities.

Generations Found	Fitness	Complexity	Introns	Dominants	Recessives
12	305.992	13	0	6	0
18	305.992	13	0	6	0
27	305.992	13	0	6	0
30	306.004	13	0	6	0
52	0.051	15	0	7	0
27	305.981	15	0	7	0
18	305.990	15	0	7	0
26	305.992	15	0	7	0
38	306.001	15	0	7	0
24	306.001	17	0	8	0
58	0.046	23	0	11	0
35	1.051	15	0	6	1
41	1.051	15	0	6	1
28	1.046	17	0	7	1
13	1.046	19	0	8	1
52	1.051	19	0	8	1
34	1.051	21	0	9	1
99	2.033	17	0	6	2
56	2.041	19	0	7	2
111	2.041	19	0	7	2
52	3.044	27	0	10	3
Median: 34	2.041	15	0	7	0

Table 32. Best-Fit Individual for Population Size of 1,000 with Probability of Mutation of 0.5

The increase in complexity is caused by the mutation operation. There is a 50% chance of replacing a mutated branch with a new full branch that can extend to a depth of between one and three. The remaining 50% extend to a depth of one to three, but cannot be complete subtrees. Which occurs depends on the random seed entered at the beginning of the run (see Appendix D). In the previous runs with the lower probabilities of mutation, only replacement by a more complex subtree during crossover could increase complexity. With the introns causing the average population fitness to fall within a smaller interval, the range of complexities also narrows. If the range of complexities is smaller, the population homogenizes because the possible combinations have been reduced. If the population is homogenous, crossover will simply exchange similar subtrees (Similar subtrees are "similar" in composition and complexity. Exchanging similar subtrees will not drive the population to a better solution).

Median Population Fitness	Median Population Complexity	Median Population Dominants	Median Population Recessives	Generation in which Recessives Disappeared from Population	Median Population Introns	Generation in which Introns Disappeared from Population
5179119	12.196	4.720	1.064	Never	0.060	Never
1688189	12.236	4.528	1.070	Never	0.062	Never
1688189	12.236	4.528	1.070	Never	0.062	Never
2413727	12.208	4.436	1.136	Never	0.064	Never
1784396	13.972	6.178	0.260	Never	0.068	Never
777749	13.980	5.278	1.158	Never	0.068	Never
1823219	12.272	5.358	0.258	Never	0.070	Never
1899637	12.268	4.516	1.108	Never	0.070	Never
1786908	12.320	4.524	1.118	Never	0.070	Never
964058	12.272	4.468	1.150	Never	0.070	Never
1934520	12.328	5.350	0.266	Never	0.072	Never
1750736	14.016	4.516	1.992	Never	0.072	Never
1970391	12.320	5.336	0.266	Never	0.074	Never
1777445	14.136	5.350	1.140	Never	0.074	Never
1945367	12.284	3.742	1.832	Never	0.074	Never
187439	16.020	5.442	2.052	Never	0.080	Never
711744.9	14.124	6.224	0.286	Never	0.082	Never
761836	14.236	6.176	0.314	Never	0.086	Never
710391	14.164	5.310	1.198	Never	0.086	Never
1597985	14.144	4.550	1.964	Never	0.088	Never
333071	22.640	4.096	6.338	Never	0.096	Never
Median: 1750736	12.328	4.72	1.118		0.072	

Table 33. Population Statistics for Population Size of 500 with Probability of Mutation of 0.5

Median Population Fitness	Median Population Complexity	Median Population Dominants	Median Population Recessives	Generation in which Recessives Disappeared from Population	Median Population Introns	Generation in which Introns Disappeared from Population
7416162	12.164	5.302	0.256	Never	0.063	Never
7253306	12.194	5.343	0.249	Never	0.064	Never
7401390	12.194	5.316	0.247	Never	0.065	Never
2475923	13.948	6.202	0.259	Never	0.068	Never
2166216	12.258	5.359	0.261	Never	0.072	Never
6619577	12.286	5.367	0.262	Never	0.072	Never
2554523	12.266	5.348	0.263	Never	0.073	Never
6860384	12.294	5.354	0.262	Never	0.074	Never
1755670	14.006	6.214	0.273	Never	0.078	Never
340829	16.340	5.403	2.061	Never	0.082	Never
1684931	14.142	5.368	1.148	Never	0.084	Never
1208383	14.180	5.328	1.195	Never	0.084	Never
909339	15.972	4.588	2.880	Never	0.084	Never
1237112	14.186	5.415	1.147	Never	0.085	Never
268274.5	17.824	5.502	2.937	Never	0.085	Never
1141135	14.148	5.379	1.158	Never	0.086	Never
1247662	14.096	6.202	0.297	Never	0.087	Never
1092638	14.170	5.422	1.145	Never	0.087	Never
1695295	14.102	5.366	1.161	Never	0.087	Never
1114348	14.586	6.236	0.307	Never	0.090	Never
345916	21.026	7.024	2.921	Never	0.091	Never
Median: 1684931	14.102	5.368	0.307		0.084	

Table 34. Population Statistics for Population Size of 1,000 with Probability of Mutation of 0.5

Generation Found	Fitness	Complexity	Introns	Dominants	Recessives
41	305.992	13	0	6	0
50	305.992	13	0	6	0
29	305.999	13	0	6	0
38	306.004	13	0	6	0
189	305.990	15	0	7	0
28	305.992	15	0	7	0
85	305.990	17	0	8	0
108	305.992	17	0	8	0
95	306.004	17	0	8	0
72	305.992	19	0	9	0
100	305.992	19	0	9	0
121	305.994	21	0	10	0
47	306.994	13	0	5	1
16	306.992	15	0	6	1
94	1.055	21	0	9	1
56	307.003	21	0	9	1
194	1.044	29	0	13	1
87	2.041	17	0	6	2
78	2.051	19	1	7	2
113	2.051	19	0	7	2
88	307.994	21	0	8	2
Median: 85	305.992	17	0	7	0

Table 35. Best-Fit Individual for Population Size of 500 with Probability of Mutation of 1.0

Generation Found	Fitness	Complexity	Introns	Dominants	Recessives
65	305.990	13	0	6	0
44	305.992	13	0	6	0
80	305.992	13	0	6	0
51	305.992	15	0	7	0
60	305.992	15	0	7	0
101	305.992	15	0	7	0
85	305.992	17	0	8	0
40	305.994	17	0	8	0
90	306.990	15	0	6	1
36	306.992	15	0	6	1
62	1.051	17	0	7	1
145	1.051	17	0	7	1
15	1.051	19	0	8	1
118	1.051	19	0	8	1
94	1.038	21	0	9	1
51	2.051	21	0	8	2
98	2.051	21	0	8	2
120	2.051	21	0	8	2
172	2.055	27	0	11	2
110	2.055	29	0	12	2
57	2.051	31	1	13	2
Median: 80	2.055	17	0	8	1

Table 36. Best-Fit Individual for Population Size of 1,000 with Probability of Mutation of 1.0

Median Population Fitness	Median Population Complexity	Median Population Dominants	Median Population Recessives	Generation in which Recessives Disappeared from Population	Median Population Introns	Generation in which Introns Disappeared from Population
369003	15.708	5.992	1.264	Never	0.242	Never
328553	17.304	4.506	3.802	Never	0.274	Never
197443	19.664	5.176	4.180	Never	0.274	Never
422245	15.996	5.254	1.692	Never	0.276	Never
332403	15.820	4.336	2.88	Never	0.284	Never
333436	17.356	6.472	1.522	Never	0.302	Never
408208	16.368	5.694	2.060	Never	0.302	Never
443893	16.620	6.236	1.618	Never	0.310	Never
334362	17.188	5.714	2.270	Never	0.320	Never
313742	17.456	6.542	1.710	Never	0.332	Never
338460	16.500	5.628	2.100	Never	0.334	Never
431152	17.204	5.462	1.738	Never	0.366	Never
388229	19.488	7.078	2.158	Never	0.372	Never
349126	19.544	7.106	1.984	Never	0.386	Never
397236	19.512	6.700	2.530	Never	0.388	Never
332500	21.060	7.932	2.494	Never	0.392	Never
376177	17.852	6.032	2.416	Never	0.408	Never
177620	25.096	7.608	4.526	Never	0.490	Never
272637	21.360	5.992	4.394	Never	0.742	Never
188207	29.144	9.044	5.714	Never	0.830	Never
249615	21.904	7.052	3.386	Never	1.062	Never
Median: 333436	17.456	6.032	2.270		0.334	

Table 37. Population Statistics for Population Size of 500 with Probability of Mutation of 1.0

Median Population Fitness	Median Population Complexity	Median Population Dominants	Median Population Recessives	Generation in which Recessives Disappeared from Population	Median Population Introns	Generation in which Introns Disappeared from Population
433491	19.07	7.095	1.852	Never	0.241	Never
417806	18.164	5.115	3.301	Never	0.242	Never
442618	17.898	6.016	2.571	Never	0.245	Never
176416	18.406	5.387	3.147	Never	0.247	Never
376612	18.034	6.274	2.256	Never	0.272	Never
353537	16.178	5.512	2.070	Never	0.275	Never
418223	17.416	6.030	2.031	Never	0.279	Never
181103	22.842	5.808	4.871	Never	0.291	Never
406136	17.814	6.915	1.531	Never	0.292	Never
396811	16.770	6.062	1.859	Never	0.301	Never
452653	22.992	6.339	3.894	Never	0.314	Never
378101	18.606	6.744	2.111	Never	0.322	Never
376331	17.204	6.112	2.078	Never	0.324	Never
366344	16.156	4.952	2.761	Never	0.332	Never
197672	24.886	5.692	5.969	Never	0.353	Never
274228	18.664	5.270	3.887	Never	0.387	Never
398919	18.022	7.242	1.261	Never	0.421	Never
381750	17.432	6.437	1.838	Never	0.421	Never
303511	22.28	5.078	5.240	Never	0.629	Never
294490	22.894	5.482	5.266	Never	0.779	Never
253455	23.650	7.848	3.359	Never	1.069	Never
Median: 376612	18.164	6.03	2.571		0.314	

Table 38. Population Statistics for Population Size of 1,000 with Probability of Mutation of 1.0

VII. CONCLUSIONS AND RECOMMENDATIONS

A. CONCLUSIONS

While an interesting study, incorporating introns and dominant/recessive genes into GPSYS-1.1, as they were implemented in this thesis, showed no improved performance except in environments featuring a high probability of mutation. Under a low probability of mutation, the population is mostly affected by the genetic drift, introduced by adding dominant and recessive genes. Introns cause the fitness to converge to a local optimum and genetic drift eventually removes the introns from the population entirely. Recessive genes are the next to succumb to the effects of genetic drift. Although dominant and recessive genes are implemented to follow Mendel's law of dominance, the population behaves as if underdominant (see Appendix C). Populations exhibiting underdominance are subject to the effects of genetic drift because they exhibit disruptive selection. In disruptive selection, the frequency of occurrence of an allele, or a function within the gene tree, is based on that allele's initial frequency. If the initial value is below a certain threshold, the allele frequency will fall to zero due to genetic drift.

For those populations that do have a lower probability of mutation, but are not completely affected by genetic drift, introns act to prevent destructive crossovers. However, because introns cause convergence, they cause the population fitness range to also fall within a smaller interval. The decreased number of different types of individuals is accompanied by a decreased chance of conducting a crossover that will produce a much higher fit offspring.

Under a high probability of mutation, the effects of genetic drift are counteracted and the population is able to achieve reasonably good results, but never as good as the unmodified version of GPSYS. The reason for the improved performance in a high mutation environment is that introns provide a subtree that can be mutated without affecting the overall fitness of the individual. Additionally, mutations that do not occur within an intron subtree are allowed to increase the complexity of the gene tree. This increase allows the gene tree to approach the minimum complexity necessary to find a good solution.

B. RECOMMENDATIONS FOR DOD

Although the results achieved in this thesis were not stellar, it does not mean that the biological mechanisms could not be implemented elsewhere or implemented differently and found successful. Chapter III provided several examples of work in which similar ideas were presented, tested, and in some cases, found successful.

C. SUGGESTED FURTHER STUDIES

If one is interested in continuing this work, the first recommendation is to reform the Chromosome class defined in GPSYS. The chromosome is haploid, but was modified in this thesis to make it act as if it were diploid. A different approach for modifying the chromosome would be to link genes together to form true homozygous and heterozygous gene pairs. These gene pairs can then be connected to form the chromosome. The difficulty of this approach is that one loses the convenient properties of a binary tree which are fully exploited in GPSYS. However, the increased complexity and the actually diploid chromosome have the potential to produce more "life-like," if not better, results.

If one is interested in applying the biological mechanisms to a different problem, we suggest that another tool, such as GALib or Avida mentioned in Chapter V, be used. These tools provide increased functionality and more choices for types of selection and types of crossover. GALib actually comes configured to make diploid chromosomes. Additionally, since they are implemented in C++, the programmer has access to pointers allowing tree manipulations to become more complex than those in GPSYS.

Something that would have helped tremendously in this research would be a compilation of the plethora of GA and GP tools available. One of the most difficult tasks of this thesis was to determine which tool to use. A worthwhile thesis would be to acquire and test the available tools, providing an overview of each tool's strengths and weaknesses.

APPENDIX A. A CELL BIOLOGY PRIMER

1. Cell Structure

It is necessary to understand the basic function of each part in the cell, collectively called organelles, in order to understand the genetic mechanisms described in this thesis.

A. The Cell Membrane

All cells have this lipid bilayer which protects the inside of the cell from the outside environment. Although the membrane is fluid and semi-permeable, it provides the outward structure of the cell.

B. The Nucleus and the Nucleolus

In cells that have a nucleus, this is where the genetic material is found. Proteins needed inside the nucleus and genetic material used outside the nucleus are transported via pores through the nuclear membrane that surrounds the nucleus. The nucleolus, which is usually visible as a dark spot in the nucleus, is the site of ribosome formation. Those organisms lacking cell nuclei are called procaryotes, whereas those that do have cell nuclei are called eucaryotes.

C. The Endoplasmic Reticulum (ER)

This organelle is the transport network for molecules targeted for certain modifications and specific final destinations, as opposed to molecules that float freely in the cytoplasm. There are two types: rough and smooth. Rough ER has ribosomes attached to it; smooth does not. The smooth portion is responsible for production of lipids, such as those used to maintain the cell membrane.

D. The Ribosomes

The organelles are responsible for protein synthesis for the cell. They can be attached to the ER, or float freely in the cytoplasm.

E. The Golgi Apparatus or Golgi Complex

This organelle sorts newly made proteins, packages them into small membrane-bound sacs called vesicles, and sends them to their proper places, both within and external from the cell. Proteins that are secreted often act as messengers to the membranes of other cells.

F. The Mitochondria

These organelles are the sites of aerobic respiration and are responsible for making adenosine triphosphate (ATP), which is the cellular energy source.

G. The Lysosomes

These organelles are small sacks which contain many enzymes. They degrade waste materials and food within the cell by breaking down molecules of DNA, protein and lipids into their base components. These base components can then be reused by the cell.

H. The Peroxisomes

The organelles that are the subcellular location of important metabolic reactions. They rid the cell of toxic substances. Relative to other organelles, little is known about how cells maintain and propagate peroxisomes.

I. The Centrioles and Vacuoles

These organelles are only found in plant cells.

J. The Cytoplasm

The cytoplasm includes everything inside the cell membrane, except the nucleus, and a fluid called the cytosol. Proteins travel around the cell through the cytosol guided by a network of fibers called the cytoskeleton. The cytoskeleton determines the shape of the cell and helps it to move.

2. Cell Division

Cellular division occurs by a process called **mitosis** (see Figure 20). The result of mitosis is the production of at least two daughter cells, each of which contains the complete copy of the chromosomes within its nucleus.

Sexually reproducing organisms also perform a special type of cell division, called **meiosis**, to create gametes (see Appendix B, Figure 22). Sexually reproducing organisms are diploid, whereas their gametes are haploid. Meiosis reduces the chromosome number in the gamete such that when the gametes from each parent fuse together, a diploid zygote is formed.

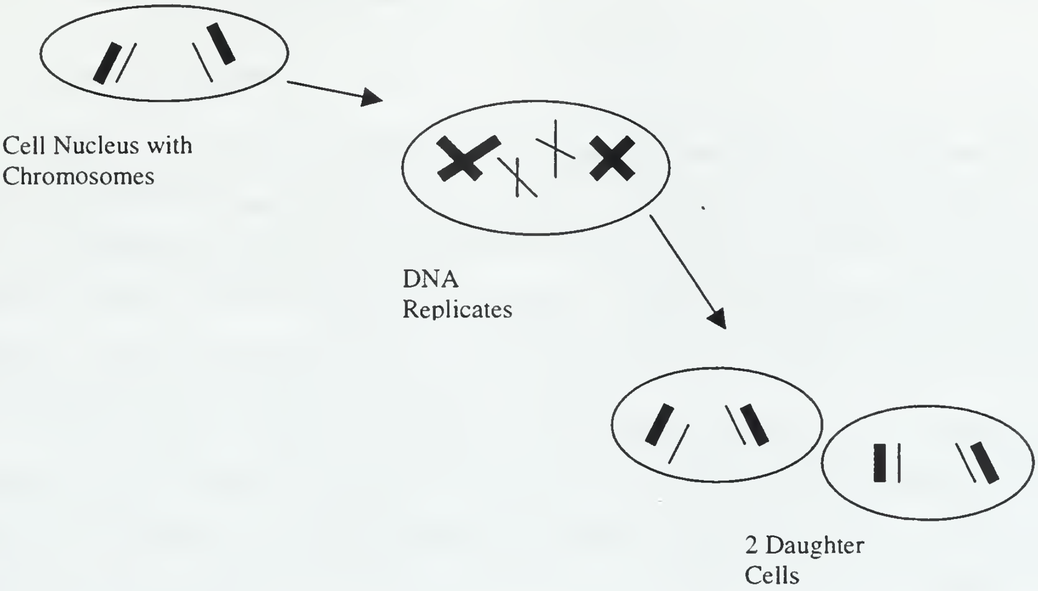


Figure 20. Mitosis

APPENDIX B. AN ADDITIONAL MOLECULAR GENETICS PRIMER

The molecular genetics primer provided in Chapter 2 was only an overview of necessary information. This appendix is provided for the reader who would like to know, or perhaps needs, additional information.

1. DNA Structure

DNA is a very long, threadlike macromolecule made up of a large number of deoxyribonucleotides, each composed of a base, a sugar and a phosphate group. The sugar and phosphate groups are responsible for the physical structure of the DNA polymer; the base carries the genetic information. These nucleotide bases fall into two categories: purines and pyrimidines. The purines in DNA are adenine (A) and guanine (G); the pyrimidines are thymine (T) and cytosine (C).

In 1953, James Watson and Francis Crick deduced the structure of DNA, the well-known double helix[24]. One of the most important aspects of the double helix is the specificity with which the nucleotide bases are paired: T only pairs with A, and G only pairs with C. As a result of that strict pairing, one strand of the double helix is the complement of the other.

2. DNA Replication

When a cell undergoes mitosis (see Appendix A), prior to the actual cell division, the parent duplicates its chromosomes by a process called DNA replication (see Figure 21). It is important to note that replication is a semi-conservative function, meaning that of the two strands found in the daughter molecule, one will be directly from the parent and the other will be newly synthesized. Assuming no error, the complementary chains acts as templates for each other during the replication. A DNA strand that can replicate is called a replicon.

[24] There are three forms to the double helix, each of which forms A-DNA, B-DNA, and Z-DNA respectively. The differences among the three will not be discussed. DNA can also be circular and supercoiled; the significance of those forms will also not be covered. Its structure does affect its chemical reactivity, so it is important to mention that there are differences.

Replication begins at a unique site at which the DNA molecule is cleaved. The highly specific enzymes used in this cleavage process recognize a palindromic nucleotide sequence that possesses two-fold rotational symmetry. In other words, if one strand of DNA has the sequence G-A-A-T-T-C, with a cleavage site between the Ts, the complementary strand will have the sequence C-T-T-A-A-G with the cleavage also between the Ts.

Replication is not completely error-free. The affinity of A for T, and G for C helps to keep copy errors low. Some bacteria have spontaneous mutation rates of 10^{-7} per replication, with the more complex fruit fly having a rate of 10^{-10} [Ref 26]. Simplifying greatly, if an incorrect base is connected to the DNA oligomer during replication, a correction enzyme will usually catch it and remove it before the next base is added. However, if an error does occur, it is classified as one of the following three mutations: substitution, insertion or deletion.

Substitution is the replacement of one base pair for another. Replacement of a purine by a purine or a pyrimidine by a pyrimidine is called a transition. The replacement of a purine by a pyrimidine (or vice versa) called a transversion. Substitution is the most common type of mutation because it is not always caught by the correction enzyme, especially when a tautomer of the base is actually added [25][Ref 27].

Insertion and deletion are simply the process of adding or deleting one or more base pairs from the resulting DNA strand. Because of the error-correcting enzymes used, these mutations are less common during replication. After the DNA is formed, though, there are types of molecules capable of intercalating between adjacent base pairs. This can lead to an insertion or a deletion. If not done in multiples of three, the effect of these mutations is to alter the reading frame of the DNA.

[25] A tautomer is any molecule that has an isomer with which it is in chemical equilibrium.

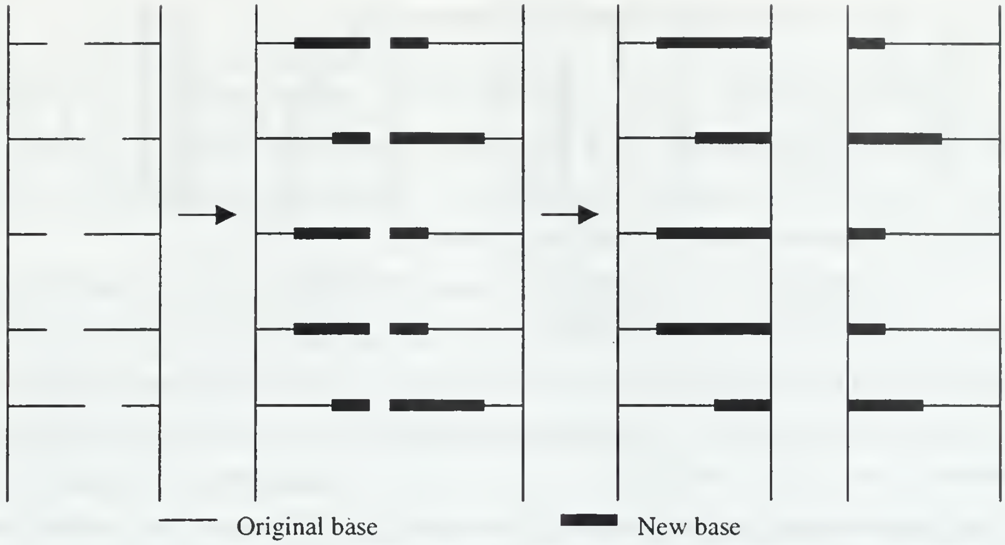


Figure 21. DNA Replication

There are a number of reasons to consider the simplest type of mutation, the exchange of one base for another. First, base switches reflect the basic accuracy of DNA replication. Second, many important mutagens act by making single-base changes. Finally, single-base mutations are critical to evolution, because they change genes in ways that are subtle enough to yield useful variants. A second important fact is that not all sites undergo mutation at the same rate; some are hot spots, at which mutations occur much more often than at most other sites.

Single-base switches are usually reversible, and often the rate of “back” mutation to the normal nucleotide arrangement is similar in order of magnitude to the rate of change to the mutant arrangement. This fact represents an important way to distinguish base switches from more drastic alterations like large deletions, for which the reverse reaction (called reversion) is impossible. An intermediate case is that of single-base insertions or deletions; these may revert, but much less often than single-base switches. Most spontaneous single-base exchanges are simply rare failures in the replication process, arising when a nucleotide is added to the growing chain even though the nucleotide does not pair normally with the template base.

3. DNA Recombination and Transposons

During meoisis, DNA is broken apart and reformed in a process called recombination (see Figure 22). If the chromosomes break during the process, each may swap a portion of its genetic material for the matching portion from its mate. This swapping of genetic material is called a crossover.

Recombination occurs outside of meoisis because it is the method by which DNA repairs itself. In fact, recombination is initiated by breaks or gaps in the DNA. During meiosis, the DNA is cut enzymatically in order to initiate the process. Regardless of why it occurs, it is important to note that the breakage points in the recombinants must lie between the same nucleotides in the two homologous chromatids, otherwise recombination would generate new DNA molecules differing in length from the parental molecules. Complementary base-pairing between DNA strands unwound from two different chromosomes puts the chromosomes in exact register, so that when crossover occurs, it occurs between identical, or nearly identical, DNA sequences. Usually these sequences are two equivalent regions of homologous chromosomes, but crossover can also occur between homologous segments in nonequivalent regions (as long as the recombinant survives). Such unequal crossing can cause duplications, insertions and deletions in chromosomes. Crossing over at nonreciprocal sites is called gene conversion.

DNA segments can also be moved by site-specific recombination, which is not guided by homology but can cut and rejoin DNA at precise sites. In general, site-specific recombination serves some type of regulatory function, such as the formation of antibody genes. Site-specific recombination has one of two consequences, depending on how the interacting sites are oriented. Recombination can either remove the intervening segment, or invert it, which will change how the gene is expressed.

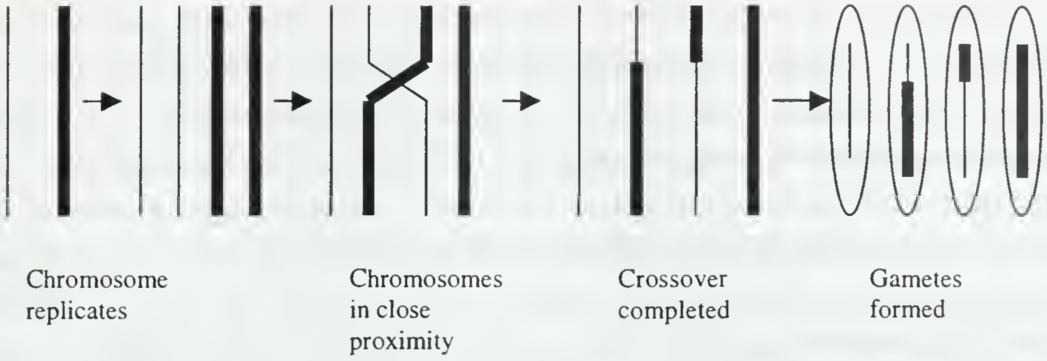


Figure 22. Meiosis

Although the existence of site-specific recombination can invert or remove DNA segments, site-specific recombination usually does not affect gene order. That is because site-specific recombinations are limited and conservative, and in higher organisms often are not inherited because they occur only in somatic cells. Transposons are different because they move genes to new and unrelated sites. They are key factors in drug-resistant bacteria.

The transposon encodes within its sequence the genes that allow it to move. There are two types: complex and simple. Complex transposons hold one or more genes in addition to the ones coding for their transposition. Simple transposons, also known as insertion sequences, only carry the genes necessary for their own transposition. Whether complex or simple, transposons interrupt or deactivate genes into which they are inserted. Transposons are identified by a sequence at one end of the gene that is repeated nearly identically, but oriented oppositely, at the other end. These areas are called inverted repeats. Although their numbers are very small, they exist in all organisms.

4. Central Dogma of Molecular Biology

Figure 1 in Chapter II is better known as the Central Dogma of Molecular Biology because it demonstrates how the sequence of a strand of DNA corresponds to the amino acid sequence of a protein. RNA was the original genetic material, but through evolution, it became an intermediary between DNA and the protein it encodes in eucaryotic organisms.

There are actually four different types of RNA that are used in gene expression: messenger RNA (mRNA), transfer RNA (tRNA), ribosomal RNA (rRNA), and small nuclear RNA (snRNA). mRNA is the template for protein synthesis. An mRNA molecule is produced for each gene or group of genes that is to be expressed. It is created from a DNA template, with a double-stranded template being the preferred type. tRNA is a small RNA that has a very specific secondary and tertiary structure such that it can bind an amino acid at one end, and mRNA at the other end. It acts as an adaptor to carry the amino acid elements of a protein to the appropriate place as coded for by the mRNA. rRNA is one of the structural components of the ribosome. Parts of its sequence are complementary to regions of the mRNA so that the ribosome knows where to bind to an mRNA from which to make a protein. snRNA is involved in the machinery that processes RNA's as they travel between the nucleus and the cytoplasm. RNA does not use the nucleotide base thymine, but one called uracil (U).

The enzyme that is used to start transcription has to find the beginning of a gene so that it knows from where to start. There is a particular DNA sequence at the beginning of genes, called a **promoter**, that is recognized by this enzyme. It is a unidirectional sequence on one strand of the DNA that tells the enzyme both where to start and on which strand to continue. Two common procaryotic promoter consensus sequences are TATAAT and TTGACA; common eucaryotic protoers are the CAAT and TATAAA boxes. Both sequences are found at specific locations on the DNA before the precise spot that mRNA is to begin. The DNA strand from which the mRNA is copied is the antisense or template strand. The other strand, to which the antisense strand is identical, is the sense or coding strand.

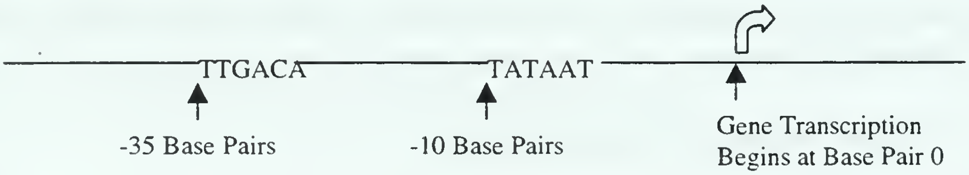


Figure 23. Promoter Consensus Sequences

Transcription is stopped when the mRNA reaches a termination signal. In general, the termination signal will be a series of G and C bases that will bind to each other in a hairpin loop that is followed by a string of U residues. The actual strand of mRNA produced is complementary to the strand from which it was synthesized and identical to the other.

(5' -> 3') ATGGAATTCTCGCTC (Coding, sense strand)
(3' <- 5') TACCTTAAGAGCGAG (Template, antisense strand)
(5' -> 3') AUGGAAUUCUCGCUC (mRNA made from Template strand)

Figure 24. Sense and Antisense Strands

If the area to be transcribed contained introns, those areas are excised out and the remaining sequences are spliced together. Introns nearly always begin with GU and end with an AG that is preceded by a pyrimidine-rich tract. This consensus sequence is part of the signal for splicing. Once the introns have been removed, the mature mRNA can leave the nucleus, and head to the ribosome where the actual protein is synthesized.

Once the mRNA binds to the rRNA of the ribosome, the tRNA can begin to build the protein. The start signal is the codon AUG found closest to the starting end, with GUG as a less common substitute. AUG codes for methionine, but this first amino acid is modified slightly. There are tRNAs for each of the amino acids, and each molecule contains an amino acid attachment site and a template-recognition site. The template-recognition site is called an anticodon, which recognizes the codon on the mRNA. The anticodon aligns with the codon, putting the amino acid in proximity to form peptide bonds with its neighbors. Translation is completed when a stop signal is reached (UAA, UAG, and UGA). Post-translational processing of the protein will usually remove the starting methionine.

5. Amino Acids and Proteins

It is important to point out that the location of each specific amino acid placement in the protein is exactly correlated with its location along the DNA. This property is called colinearity. Stated simply, colinearity is the property by which successive amino acids in a polypeptide chain are coded for by successive regions of a gene.

There are twenty amino acids used in protein synthesis. These amino acids are divided into two groups, essential and non-essential. Essential amino acids must be ingested in order to maintain health, whereas non-essential amino acids are produced within the body (see Figure 25).

AMINOACID	*SIGNIFICANT FUNCTION*
Alanine (ALA)	Non-essential. Found in brain, and central nervous system tissue
Arginine (ARG)	Non-essential. Crucial for optimal muscle growth and tissue repair
Asparagine (ASN)	Non-essential. Used in metabolic cycles
Aspartic acid (ASP)	Non-essential. Aids in the expulsion of ammonia from the body
Cysteine (CYS)	Non-essential. Necessary for the formation of skin
Glutamic acid (GLU)	Non-essential. Used in many brain functions
Glutamine (GLN)	Non-essential. Used to form fibrin clots
Glycine (GLY)	Non-essential. Used in making immune system hormones
Histidine (HIS)	Non-essential. Found in hemoglobin
Isoleucine (ILE)	Essential. Used to make essential biochemical components
Leucine (LEU)	Essential. Used to make essential biochemical components
Lysine (LYS)	Essential. Helps the body to absorb calcium and form collagen
Methionine (MET)	Essential. Principle supplier of sulfur to prevent disorders of the hair, skin, and nails; regulates the formation of ammonia and creates ammonia-free urine.
Phenylalanine (PHE)	Essential. Used in the brain to produce norepinephrine, a chemical that transmits signals between nerve cells
Proline (PRO)	Non-essential. Important for the functioning of joints and tendons
Serine (SER)	Non-essential. Synthesizes fatty acid sheath around nerve fibers
Threonine (THR)	Essential. An important part of collagen, Elastin, and enamel protein
Tryptophan (TRP)	Essential. Natural relaxant that helps the immune system
Tyrosine (TYR)	Non-essential . Transmits nerve impulses to the brain and promotes functioning of the thyroid, adrenal and pituitary glands
Valine (VAL)	Essential. Found in muscle tissue

Figure 25. The Amino Acids

6. RNA was the Original Molecule of Heredity

It has been shown that RNA molecules as well as proteins can be enzymes. As a result of this finding, it was proposed that RNA catalyzed its own replication and developed other enzymatic activities. DNA was formed by the reverse transcription of this RNA. DNA replaced RNA as the genetic material because its double helix is a more stable and more reliable store of genetic information [Ref 28].

Recent analyses of DNA sequences from many genomes provide hints of how such an early coding system may have been structured. The common characteristic found in today's DNA is a constancy of the relative positions of purines and pyrimidines with protein-coding regions. Specifically, codons of the form RNY predominate (where R is a purine, Y is a pyrimidine, and N can be either), suggesting that all codons may have been of this type originally. A primitive message composed exclusively of RNY codons could have been translated in only one of the three posssible frames, circumventing the need for special start signals to fix the reading frame. Interestingly, among the eight amino acids specified by RNY in today's code are amino acids that are most likely to have been generated by prebiotic synthesis, as well as those that often appear in meteorites [Ref 29].

If today's genomes are searched for RNY periodicity, the extent of coding regions and their correct reading frames can usually be identified. Genes that are well expressed seem to have best preserved the RNY pattern over their entire length. In other genes, the original RNY message can still be detected but appears badly mutated or is shifted over parts of the coding region into other reading frames by appropriate deletion-insertion pairs [Ref 30].

Why might the RNY pattern have been favored during early evolution over other possible codon patterns? RNY is a self-complementary sequence. Moreover, a repeating RNY pattern can be perceived in a master tRNA sequence compiled from the several hundred tRNAs analyzed. Thus, it has been suggested that primitive tRNAs may have served dual roles as both adaptors and mRNAs. According to this scheme, before ribosomes appeared, the entire process of gene expression could have been carried out by a single class of RNA molecules [Ref 31].

APPENDIX C. POPULATION GENETICS PRIMER

1. Review of Mendelian Genetics

Mendel proposed three laws that have been subsequently expanded, but still remain the valid rules of heredity. The law of dominance says that in a heterozygote, which is a diploid organism whose alleles are different at a specific locus, one allele may conceal the presence of another. The allele expressed in the phenotype is the dominant trait; the other is the recessive trait. A homozygote is an organism whose alleles are the same at that locus, whether they both are dominant or both recessive. It is important to point out that in some organisms the heterozygous phenotype is intermediate between the homozygous phenotypes, thereby exhibiting neither dominance nor recessiveness. The law of segregation says that in a heterozygote, two different alleles segregate from each other during gamete formation so that gametes constitute equally-proportioned single copies of the genotype of that individual. The law of independent assortment states that the expression of a gene for any single characteristic is usually not influenced by the expression of another characteristic. It has subsequently been shown that some genes on certain chromosomes exhibit non-independence, but this law is true for most cases.

2. How Alleles Differ

There is a subtle but important distinction emphasized in population genetics about the differences between alleles. Alleles are said to differ by origin if they come from the same locus on different chromosomes. Two alleles at a specified locus in a diploid individual are always different by origin. Alleles differ by state in one of two ways. If the context of the difference refers to the DNA sequence of the alleles, then the alleles are different by state if they have different DNA sequences. If the context is the product produced by the allele, the alleles differ by state if and only if they have different amino acids at a particular site. The final way alleles can differ is by descent; this occurs when they do not share a common ancestor within a certain finite number of generations.

3. The Hardy-Weinberg Law

Because it is impossible to describe the genetic structure of a real-world population by listing all the genotypes within it, relative frequencies of alleles and genotypes are used. The relative frequencies of alleles and genotypes are determined by probabilistic reasoning. For example, if a locus has two alleles, *a1* and *a2*, there are three possible genotypes: *a1a1*, *a1a2*, and *a2a2*. Each genotype occurs with relative frequency x_{11}, x_{12}, and x_{22} respectively and $x_{11} + x_{12} + x_{22} = 1$. The frequency of the *a1* allele is $x_{11} + 1/2\ x_{12}$ and that of *a2* is $1/2x_{12} + x_{22}$.

The Hardy-Weinberg law relates allele frequency and genotype frequencies at an autosomal locus in a randomly mating, infinitely sized population at equilibrium. The equilibrium state of a single locus is reached if the population is free of other evolutionary forces such as mutation, migration and genetic drift. Random mating implies that mates are chosen with complete ignorance of their genotype, degree of relationship (incest or inbreeding permissible), or geographic locality[26]. For the same genotypes above, the frequency will be p^2, $2pq$, q^2 respectively, where p is the frequency of the *a1* allele and q is the frequency of the *a2* allele. Using this law, it is easy to show that rare alleles are mostly found in heterozygotes and as a consequence their fate is tied to the dominance relationship. Thus, dominance is an important factor in evolution.

4. Genetic Drift and the Neutral Theory

In finite populations, random changes in allele frequencies result from variation in the number of offspring between individuals and, if the species is diploid and sexual, from Mendel's law of segregation. These random changes are called genetic drift. Genetic drift affects evolution in two important ways—it acts as a dispersive force that removes genetic variation and it affects the probability of survival of new mutations. Mutation counters the dispersive force by putting variation back into the population. The neutral theory claims that much molecular variation is due to the interaction of drift and mutation.

Genetic drift is an evolutionary force that changes both allele and genotype frequencies, but its effects are very weak in large populations. Roughly speaking, the

[26] A description of the genetic structure of the population must include a geographic component if the ultimate goal is to understand the evolutionary forces responsible for genetic variation.

time required for genetic drift to reduce the heterozygosity of the population by one-half is proportional to the population size. Its effects are much more pronounced on rare alleles and on small, subdivided populations with low migration rates. In the case of mutations, as long as the mutation is a neutral mutation (the end products are functionally the same), then the rate of substitution of the mutation in the population is the same as the mutation rate; the rate of substitution does not depend on the population size [Ref 32].

The neutral theory claims that most allelic variation and substitutions in proteins and DNA are neutral. It is called Non-Darwinian evolution because most substitutions are due to genetic drift, not natural selection.

5. Natural Selection and Dominance

Natural selection is the evolutionary force most responsible for adaptation to the environment. Natural selection changes allele frequencies, but only works when the genotypes have different fitnesses[27]. The dominance relationships between alleles affecting fitness thus affect the outcome of the selection.

The concept of fitness in biological organisms is typically associated with the probability that an individual will survive to adulthood in order to reproduce. The probability of survival, or the fraction of individuals that survive, will in general, depend on the genotype and this probability is better known as viability. Thinking of viability in terms of the fraction of individuals that survive to adulthood makes it easier to see that the frequency of the genotype after selection is proportional to the genotype's original frequency multiplied by its viability (see Table 39). It is also easier to compare the relative frequencies of genotypes since absolute fitness is a difficult quantity to determine and the individual fitness may include non-genotype influences.

Using the information in Table 39, the change in the allele frequency in a single generation is determined by the equation:

$$\Delta_s p = \frac{pq[p(w_{11} - w_{12}) + q(w_{12} - w_{22})]}{p^2 w_{11} + 2pq w_{12} + q^2 w_{22}}$$

[27] The common fitnesses are individual fitness, genotype fitness, relative fitness and absolute fitness.

Genotype	*a1a1*	*a1a2*	*a2a2*
Newborn Frequency	p^2	$2pq$	q^2
Viability	w_{11}	w_{12}	w_{22}
Relative Post Selection Frequency	$p^2\, w_{11}/\overline{w}$	$2pq\, w_{12}/\overline{w}$	$q^2\, w_{22}/\overline{w}$ *

* $\overline{w}$ - a constant of proportionality, known as the mean fitness of the population

Table39. Relative Post Selection Frequency

Mendel described complete dominance in which one allele clearly masked the existence of the other allele. However, incomplete dominance, overdominance and underdominance are the most interesting to evolution because most cases of complete dominance abounds for morphologic traits. The type of dominance is determined by a parameter, h, called the heterozygous effect, when comparing the relative fitness between genotypes. It is a measure of the fitness of the heterozygote relative to the selective difference between the two homozygotes (see Figure 26).

Genotype	*a1a1*	*a1a2*	*a2a2*
Relative fitness	1	$1\text{-}hs$ *	$1\text{-}s$

**s-selection coefficient*

$h = 0$ *a1* dominant, *a2* recessive

$h = 1$ *a1* recessive, *a2* dominant

$0 < h < 1$ incomplete dominance

$h < 0$ overdominance

$h > 1$ underdominance

$1\text{-}hs = w_{12}/\, w_{11}$

$1\text{-}s = w_{22}/\, w_{11}$

Figure 26. Relative Fitness of a One Locus, Two Allele Genotype

By using the relative frequencies, the change in the allele frequency can be rewritten as

$$\Delta_s p = \frac{pqs[ph + q(1-h)]}{\overline{w}},$$

and the mean fitness for the population becomes $\overline{w} = 1 - 2pqhs - q^2 s$.

The three evolutionary significant types of dominance are closely associated with the three types of natural selection. Incomplete dominance is linked with directional selection, overdominance with balancing selection, and underdominance with disruptive selection. Directional selection is the type of selection to which Darwin unknowingly referred since population genetics was not yet in existence. This type of selection implies that the fitness of *a1a1* exceeds that of *a1a2*, which in turn exceeds that of *a2a2*. This occurs with incomplete dominance ($0 < h < 1$). As a result, the frequency p is always increasing (see Figure 27). The rate by which p changes is strongly dependent on p itself. Evolution proceeds very slowly when there is little genetic variation (p is close to zero or close to one), and is most effective when $p = ½$.

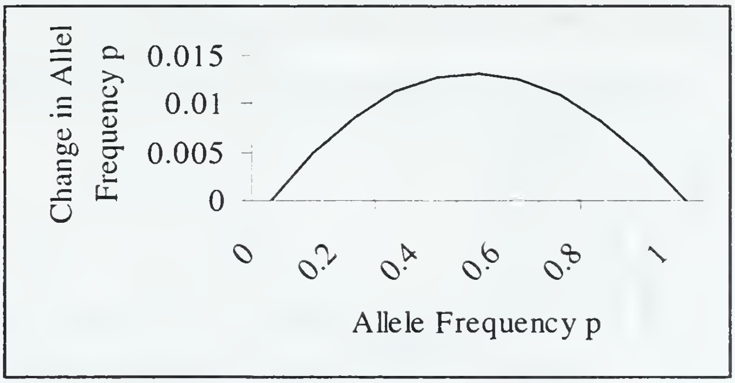


Figure 27. Change in the Allele Frequency in a Single Generation

Balanced selection occurs when there is overdominance in the allele. From Figure 28, when p is close to zero, the allele frequency will increase; when close to one, the allele frequency will decrease. There is a point in between, called the equilibrium value, at which the frequency no longer changes.

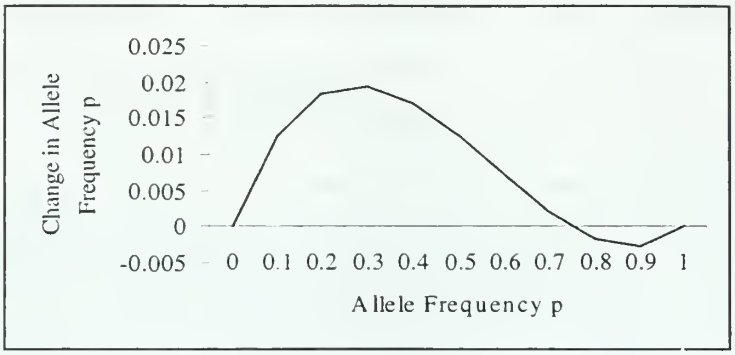


Figure 28. Balanced Selection

In disruptive selection, the allele frequency depends on the initial frequency. If the initial value is less than the equilibrium value, the frequency will approach zero. The allele frequency will approach one if it is greater than the equilibrium value. If the two are equal, there will be no change at all. A small change in *p*, such as that caused by genetic drift, will cause the allele frequency to move from the equilibrium value. This will only happen when there is underdominance (see Figure 29).

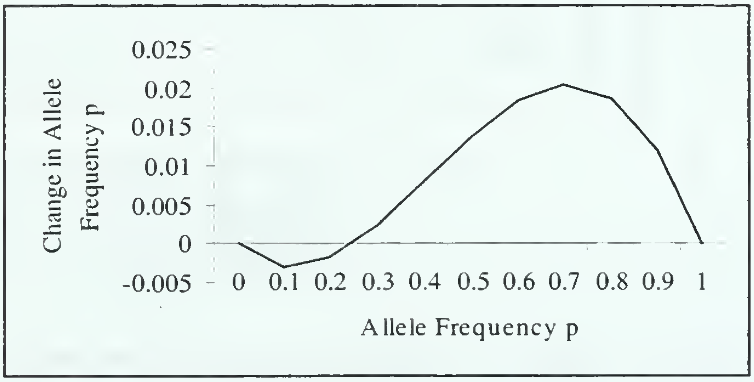


Figure 29. Disruptive Selection

APPENDIX D. MODIFIED GPSYS-1.1 CODE AND NEWLY WRITTEN CODE

1. CHROMOSOME

```
/*
      Copyright Adil Qureshi - 1997
      This code is part of gpsys Release 1.1
      and is released for non-commercial use only.
      Questions, comments etc should be forwarded to :-

            Adil Qureshi
            University College London,
            Department of Computer Science,
            Gower St,
            London WC1E 6BT, UK.
            Tel: +44 (0)171 380 7777 x4436
            Fax: +44 (0)171 387 1397
            email: A.Qureshi@cs.ucl.ac.uk
            URL : http://www.cs.ucl.ac.uk/staff/A.Qureshi/

      Modified by Captain Loretta Vandenberg
            Naval Postgraduate School
            Monterey, CA  93943
      All changes are marked.  Some additional comments
      Added to the original code in order to clarify
      Usage.
*/

package gpsys;

/**
 * A Chromosome defines an evolvable gene tree.
 *
 * @see gpsys.Terminal
 *
 * @version       1.1, 30th June '97
 * @author <a href="mailto:A.Qureshi@cs.ucl.ac.uk">Adil Qureshi</a>
 *       <address>Department of Computer Science,</address>
 *       <address>University College London,</address>
 *       <address>Gower St,</address>
 *       <address>London WC1E 6BT,</address>
 *       <address>UK.</address>
 *
 * @author <a href="mailto:shirley@cs.nps.navy.mil">Rett Vandenberg</a>
 *       <address>Department of Computer Science</address>
 *       <address>Naval Postgraduate School</address>
 *       <address>Monterey, CA  93943</address>
 *
 */
```

```
public class Chromosome implements Cloneable, java.io.Serializable  {

   /**
    * The Gene at the top of the tree.
   * this gene is evaulated when this ADF needs evaluation
   *
   * @see Gene
   */
   public Gene treeTop;

   /**
    * The GPParameters used to create this chromosome
    *
    * @see GPParameters
    */
   public   GPParameters gpParameters;

   /**
    * index into the adf array in the gpParameters.  It is used to
    *acces the ChromosomeParameters associated with this chromosome.
    */
   int adf;

   /**
    * A count of the total number of nodes in this tree.
    */
   int complexity;

   /**
    * Added by Rett Vandenberg
    * A count of the total number of introns in this tree.
    */
   int introns;

   /**
    * Added by Rett Vandenberg
    * A count of the total number of dominants in this tree.
    */
   int dominants;

   /**
    * Added by Rett Vandenberg
    * A count of the total number of recessives in this tree
    */
   int recessives;
```

```
/**
 * Constructs a new Chromosome using the specified
 * GPParameters.
 *
 * @param      p           the GPParameters to use.
 * @param      adfIndex    the index into the adf array
 *                   in GPParameters the latter
 *                         defines ChromosomeParameters
 *                   for each adf.
 * @exception TypeException      If there was a typing problem
 *                   during tree generation.  For
 *                         example a Function or Terminal
 *                         of the required type could not
 *                         be found.
 */
Chromosome(GPParameters p, int adfIndex) throws TypeException {
   gpParameters = p;
   adf = adfIndex;
   // always start the tree with a function
   int createMethod = p.adf[adfIndex].createMethod;
   if (createMethod == ChromosomeParameters.CREATE_GROW){
      treeTop = new
         GeneFunctionGrow(
            p.adf[adfIndex].maxDepthAtCreation - 1,
                  p.adf[adfIndex].type, p, adfIndex);
   }else if (createMethod == ChromosomeParameters.CREATE_FULL){
      treeTop = new
         GeneFunctionFull(
            p.adf[adfIndex].maxDepthAtCreation - 1,
                p.adf[adfIndex].type, p, adfIndex);
   }else { // assume (createMethod == CREATE_RAMP_HALF_AND_HALF)
      // ramp up the depth;
      //minimum = 2, maximum = maxDepthAtCreation
      double depthValue =
         p.adf[adfIndex].maxDepthAtCreation *
         (p.creationIndex / p.populationSize);
      int maxDepth =
         (depthValue < 2) ? 2 : (int) depthValue;

      // half the population created via grow,
      // and the other via full
      if ((p.creationIndex % 2) == 0){
         treeTop = new GeneFunctionGrow(maxDepth,
               p.adf[adfIndex].type, p, adfIndex);
      }else {
         treeTop = new GeneFunctionFull(maxDepth,
               p.adf[adfIndex].type, p, adfIndex);
      }
   }

   complexity = treeTop.complexity();
   //added by Rett Vandenberg
   introns = treeTop.introns();
   dominants = treeTop.dominants();
```

```
      recessives = treeTop.recessives();
}
/**
 * Creates a new child Chromosome which is a mutation of the
 * mother.
 *
 * @param   mum   is the mother Chromosome.
 * @return  a reference to a child Chromosome which is a
 *          mutation of the mother
 *
 */
public static Chromosome mutate(Chromosome mum) {

   // a convenience variable
   GPParameters gpParameters = mum.gpParameters;

      // child is initially a clone of it's mum
      Chromosome child = mum.deepClone();

      // pick a branch at random from the child
      GeneBranch branch = new
         GeneBranch(gpParameters.rng, child.treeTop);

      // just for convenience
      int depth =
         gpParameters.adf[child.adf].maxDepthMutation - 1;

      // now generate the mutant branch using FULL or
      //GROW methods
      Gene newBranch;
         if ((gpParameters.rng.nextInt() % 2) == 0) {
         // use FULL method
            try {
               newBranch = new
                  GeneFunctionFull(depth,
                              branch.child.p.type,
                              gpParameters, child.adf);
            }
            catch (TypeException e) {
               newBranch = null;
            }
         } else {
            // use GROW method
            try {
               newBranch = new
                  GeneFunctionGrow(depth,
                              branch.child.p.type,
                              gpParameters, child.adf);
            }
            catch (TypeException e) {
               newBranch = null;
            }
         }
      if (newBranch == null) {
         try {
            newBranch = new GeneTerminal(depth,
                              branch.child.p.type,
                              gpParameters, child.adf);
         }
         catch (TypeException e) {
            return child;
         }
      }

      // if the root of child's tree was mutated, replace
      // the whole tree
      if (branch.parent == null) {
         child.treeTop = newBranch;
      }
      else {
         // save the branch to be mutated
         Gene tmp =
       ((GeneFunction) branch.parent).arguments[branch.index];

       // update the tree to include the mutated branch
      ((GeneFunction) branch.parent).arguments[branch.index] =
                                                      newBranch;

      // if the mutated tree is too big, child == mum
      if (child.treeTop.depth() > gpParameters.adf[child.adf].maxDepth)
      {
         ((GeneFunction) branch.parent).arguments[branch.index] = tmp;
         gpParameters.observer.diagnosticUpdate(
            "Throwing away a tree after mutation");
      }
   }

   child.complexity = child.treeTop.complexity();

   //added by Rett Vandenberg
   child.introns = child.treeTop.introns();
   child.dominants = child.treeTop.dominants();
   child.recessives = child.treeTop.recessives();

   return child;
}


/**
 * Creates a new child Chromosome via crossover of the mother
 * and father Chromosomes.
 *
 * @param   mum is the mother Chromosome.  The child is
 *          actually a copy of the mother with one branch
 *          exchanged with a branch from the father.
 * @param   dad is the father Chromosome.
 * @return  a reference to a new child Chromosome.
 */
```

```
public static Chromosome cross(Chromosome mum, Chromosome dad){

   // a convenience variable
   GPParameters gpParameters = mum.gpParameters;




   //this does not prevent it, it just warns you it is happening
   //I never saw it occur
   if (mum == dad) {
      gpParameters.observer.diagnosticUpdate(
                        "Danger - XOVER is incestious");
   }

   // make the child a clone of mum
   Chromosome child = mum.deepClone();

   // pick a brach at random from the child to replace
   GeneBranch branchMum =
               new GeneBranch(gpParameters.rng, child.treeTop);

   // pick a branch from dad to replace it with
   // the new branch must return the same type as the
   // mum branch
   GeneBranch branchDad = new
               GeneBranch(gpParameters.rng, dad.treeTop,
                          branchMum.child.p.type);

   // if now such branch found in dad, child == mum
   if (branchDad.child == null) {
      gpParameters.observer.diagnosticUpdate(
        "Couldn't find compatible branch in dad during crossover");
      return child;
   }

   // make a copy of dad's branch
   Gene newBranch = branchDad.child.deepClone();

   // if replacing root of the child tree, tree = copy
   // of dad's branch
   if (branchMum.parent == null) child.treeTop = newBranch;
   else {
      // save the branch being replaced
      Gene tmp = ((GeneFunction) branchMum.parent).
                        arguments[branchMum.index];

      // replace the branch with one from dad
      ((GeneFunction) branchMum.parent).
                  arguments[branchMum.index] = newBranch;

      //Added by Rett Vandenberg
      //dominant/recessive test
      //Based on Mendelian heredity
   //double dominant parents make a double dominant child
   if (tmp.allele == 2 && newBranch.allele == 2) {
      child.treeTop.allele = 2;

   //One double dominant and One double recessive can only
   //produce heterozygous kids
   } else if (tmp.allele == 2 && newBranch.allele == 0) {
      child.treeTop.allele = 1;
   } else if (tmp.allele == 0 && newBranch.allele == 2) {
     child.treeTop.allele = 1;

   //two double recessives can only produce recessive child
   } else if (tmp.allele == 0 && newBranch.allele == 0) {
     child.treeTop.allele = 0;

   //if you have a heterozygous parent, and a double recessive
   //parent, you have a 50% chance of a heterozygous child
   //and a 50% chance for a double recessive child
   } else if ((tmp.allele == 1 && newBranch.allele == 0) ||
              (tmp.allele == 0 && newBranch.allele == 1)) {
         //this allows a 50% chance based on the
         //random seed provided
         if ((gpParameters.rng.nextInt() % 2) == 0) {
            child.treeTop.allele = 1;
         } else {child.treeTop.allele = 0;}

   //a double dominant parent with a heterozygous parent
   } else if ((tmp.allele == 2 && newBranch.allele == 1) ||
              (tmp.allele == 1 && newBranch.allele == 2)) {
         //50% chance of double dominant or heterozygous
         if ((gpParameters.rng.nextInt() % 2) == 0) {
            child.treeTop.allele = 2;
         } else {child.treeTop.allele = 1;}
   } else {
         //both parents are heterozygous
   //there's  50% chance of a heterozygous child
   if ((gpParameters.rng.nextInt() % 2) == 0) {
      child.treeTop.allele = 1;

   //a 25% chance of a double dominant
   } else if ((gpParameters.rng.nextInt() % 4) == 0) {
      child.treeTop.allele = 2;

   //technically, a 25% chance for a double recessive, but
   //one cannot implement the ambiguity, so this may or may
   //not ever achieve a 25%
   } else {child.treeTop.allele = 0;}
}

// if the resulting tree is too big, child == mum
if (child.treeTop.depth() > gpParameters.adf[child.adf].maxDepth) {
   ((GeneFunction) branchMum.parent).arguments[branchMum.index] =
                           tmp;
   gpParameters.observer.diagnosticUpdate(
                        "Throwing away a tree after Xover");
   }
}
child.complexity = child.treeTop.complexity();

//added by Rett Vandenberg
child.introns = child.treeTop.introns();
child.dominants = child.treeTop.dominants();
child.recessives = child.treeTop.recessives();
return child;
}//end cross


/**
 * Makes a deep copy of this Chromsome by making a copy of the entire
 * data graph.
 *
 * @return  a reference to a copy of this Chromosome.
 */
public Chromosome deepClone() {
   Chromosome clone = null;
   try {
      clone = (Chromosome) this.clone();
      clone.treeTop = this.treeTop.deepClone();
   }
   catch (CloneNotSupportedException e) {
      // will never happen as long as we implement Cloneable
   }
   return clone;
}

/**
 * Calculates the number of nodes in the Gene tree of this Chromosome.
 *
 * @return  the number of nodes in the Gene tree.
 */
public int complexity() {return complexity;}

/**
 * Added by Rett Vandenberg
 * Calculates the number of introns in the Gene tree of this
 * Chromosome.
 *
 * @return  the number of introns in the Gene tree.
 */
public int introns() {return introns;}

/**
 * Added by Rett Vandenberg
 * Calculates the number of dominants in the Gene tree of
 * this Chromosome.
 *
 * @return  the number of introns in the Gene tree.
 */
```

```
public int dominants() {return dominants;}

/**
 * Added by Rett Vandenberg
 * Calculates the number of introns in the Gene tree of this
 * Chromosome.
 *
 * @return  the number of introns in the Gene tree.
 */
public int recessives() {return recessives;}

/**
 * Evaluates the chromosome so that it returns an Object
 * reference.
 *
 * @param   i is the Individual being evaluated.
 * @return  a reference to an Object returned by evaluating the
 * Gene tree.
 * @exception     EvaluationException     If there is an
 * evaluation failure.
 */
public Object evaluateObject(Individual i) throws
   EvaluationException {return treeTop.evaluateObject(i);}

/**
 * Evaluates the chromosome so that it returns a byte.
 *
 * @param   i is the Individual being evaluated.
 * @return  a byte returned by evaluating the Gene tree.
 * @exception     EvaluationException     If there is an
 * evaluation failure.
 */
public byte evaluateByte(Individual i) throws
   EvaluationException {return treeTop.evaluateByte(i);}

/**
 * Evaluates the chromosome so that it returns a byte.
 *
 * @param   i is the Individual being evaluated.
 * @return  a byte returned by evaluating the Gene tree.
 * @exception     EvaluationException     If there is an
 * evaluation failure.
 */
public short evaluateShort(Individual i) throws
   EvaluationException {return treeTop.evaluateShort(i);}

/**
 * Evaluates the chromosome so that it returns an int.
 *
 * @param   i     is the Individual being evaluated.
 * @return  an    int returned by evaluating the Gene tree.
 * @exception     EvaluationException     If there is an
 * evaluation failure.
 */
```

```
public int evaluateInt(Individual i) throws
   EvaluationException {return treeTop.evaluateInt(i);}

/**
 * Evaluates the chromosome so that it returns a long.
 *
 * @param   i is the Individual being evaluated.
 * @return  a long returned by evaluating the Gene tree.
 * @exception     EvaluationException     If there is an
 * evaluation failure.
 */
public long evaluateLong(Individual i) throws
   EvaluationException {return treeTop.evaluateLong(i);}

/**
 * Evaluates the chromosome so that it returns a float.
 *
 * @param   i is the Individual being evaluated.
 * @return  a float returned by evaluating the Gene tree.
 * @exception     EvaluationException     If there is an
 * evaluation failure.
 */
public float evaluateFloat(Individual i) throws
   EvaluationException {return treeTop.evaluateFloat(i);}

/**
 * Evaluates the chromosome so that it returns a double.
 *
 * @param   i is the Individual being evaluated.
 * @return  a double returned by evaluating the Gene tree.
 * @exception     EvaluationException     If there is an
 * evaluation failure.
 */
public double evaluateDouble(Individual i) throws
   EvaluationException {return treeTop.evaluateDouble(i);}

/**
 * Evaluates the chromosome so that it returns a char.
 *
 * @param   i is the Individual being evaluated.
 * @return  a char returned by evaluating the Gene tree.
 * @exception     EvaluationException     If there is an
 * evaluation failure.
 */
public char evaluateChar(Individual i) throws
   EvaluationException {return treeTop.evaluateChar(i);}

/**
 * Evaluates the chromosome so that it returns a boolean.
 *
 * @param   i is the Individual being evaluated.
 * @return  a boolean returned by evaluating the Gene tree.
 * @exception     EvaluationException     If there is an
 * evaluation failure.
 */
public boolean evaluateBoolean(Individual i) throws
   EvaluationException {return treeTop.evaluateBoolean(i);}

/**
 * Generates a String representing  a dump of the Gene tree for
 * this Chromosome.
 *
 * @return  a String representing the Gene tree for this Chromosome.
 */
public String toString() {return treeTop.toString();}

}//end Chromosome
```

2. GENE

```
/*
      Copyright Adil Qureshi - 1997
      This code is part of gpsys Release 1.1
      and is released for non-commercial use only.
      Questions, comments etc should be forwarded to :-

            Adil Qureshi
            University College London,
            Department of Computer Science,
            Gower St,
            London WC1E 6BT, UK.
            Tel: +44 (0)171 380 7777 x4436
            Fax: +44 (0)171 387 1397
            email: A.Qureshi@cs.ucl.ac.uk
            URL : http://www.cs.ucl.ac.uk/staff/A.Qureshi/

      Modified by Captain Loretta Vandenberg

            Department of Computer Science
            Naval Postgraduate School
            Monterey, CA  93943

*/

package gpsys;

/**
 * A Gene is a node in a GP tree.  A Gene can be either a function or a
 * terminal, hence this abstract class has been subclassed to
 * GeneFunction and GeneTerminal which are actually used.
 *
 * @version       1.1, 30th June '97
 * @author <a href="mailto:A.Qureshi@cs.ucl.ac.uk">Adil Qureshi</a>
 *       <address>Department of Computer Science,</address>
 *       <address>University College London,</address>
 *       <address>Gower St,</address>
 *       <address>London WC1E 6BT,</address>
 *       <address>UK.</address>
 *
 * @author <a href="mailto:shirley@cs.nps.navy.mil">Rett Vandenberg</a>
 *       <address>Department of Computer Science</address>
 *       <address>Naval Postgraduate School</address>
 *       <address>Monterey, CA  93943</address>
 */
public abstract class Gene implements java.io.Serializable {

   /**
    * The primitive associated with this Gene which is either a
    * Function or a Terminal.
    */
    public Primitive p;

      /**
       * Added by Rett Vandenberg
       * A flag to determine whether or not this particular gene is
       * an intron or not.  It will always be set to false for
       * terminals, and false for non-intron function genes
       */
      public boolean intron;

      /**
       * Added by Rett Vandenberg
       * A flag to determine whether or not this particular gene is
       * a dominant or recessive.  It will always be set to false for
       * terminals intron function genes.  Recessive functions will
       * be set to true.  Dominant functions are those provided.
       */
      public boolean dominant;

      /**
       * Added by Rett Vandenberg
       * An integer to determine whether or not this particular gene is
       * a dominant or recessive.  Two is assigned for double
dominants,
       * 1 for heterozygous dominant, and 0 for double recessive.
       * Initial population is assigned values at random.  Checked
       * checked during cross (see Chromosome) to determine the value
       * given to offspring.
       */
      public int allele;

      /**
       * Calculates the number of nodes (Genes) in the subtree
       * starting at this node (Gene).
       */
      abstract public int complexity();

      /**
       * Calculates the maximum depth of the subtree starting at
       * this node (Gene).
       */
```

```
  abstract public int depth();

  /**
   * Added by Rett Vandenberg
   * An abstract function call that will determine the number
   * of introns in a given sub-branch.  It must be instantiated
   * by the GeneFunction class
   */
  abstract public int introns();

  /**
   * Added by Rett Vandenberg
   * An abstract function that will determine the number of
   * dominants in a given sub-branch.
   * It must be instantiated
   * by the GeneFunction class
   */
  abstract public int dominants();

 /**
  * Added by Rett Vandenberg
  * An abstract function that will determine the number of
  * recessives in a given sub-branch.
  * It must be instantiated
  * by the GeneFunction class
  */
 abstract public int recessives();

/**
 * Added by Rett Vandenberg
 * Intron test.  Used in Fitness evaluation
 */
 public boolean isIntron() {
    if (this.intron == true) {return true;}
    else {return false;}
 }

/**
 * Added by Rett Vandenberg
 * Dominant test.  Used in Fitness evaluation
 */
 public boolean isDominant() {
    if (this.dominant == true) { return true; }
    else {return false;}
 }

/**
 * Added by Rett Vandenberg
 * Recessive test.  Used in Fitness evaluation
 */
 public boolean isRecessive() {
    if (this.dominant == false) { return true; }
    else {return false;}
 }
```

```
    /**
     * Evaluate this Gene as a Gene that returns an Object when
     * evaluated.
     *
     * @param     i     the Individual being evaluated.
     * @return    An Object which is the evaluation result.
     * @exception EvaluationException     If there is an evaluation
     *                                    failure.
     */
     public abstract Object evaluateObject(Individual i)
            throws EvaluationException;

    /**
     * Evaluate this Gene as a Gene that returns a byte when evaluated.
     *
     * @param     i     the Individual being evaluated.
     * @return    A byte which is the evaluation result.
     * @exception EvaluationException     If there is an evaluation
     *                                      failure.
     */
     public abstract byte evaluateByte(Individual i)
            throws EvaluationException;

    /**
     * Evaluate this Gene as a Gene that returns a short when
evaluated.
     *
     * @param     i     the Individual being evaluated.
     * @return    A short which is the evaluation result.
     * @exception EvaluationException     If there is an evaluation
     *                                    failure.
     */
     public abstract short evaluateShort(Individual i)
            throws EvaluationException;

    /**
     * Evaluate this Gene as a Gene that returns a int when evaluated.
     *
     * @param     i     the Individual being evaluated.
     * @return    A int which is the evaluation result.
     * @exception EvaluationException     If there is an evaluation
     *                                    failure.
     */
     public abstract int evaluateInt(Individual i)
            throws EvaluationException;

    /**
     * Evaluate this Gene as a Gene that returns a long when evaluated.
     *
     * @param     i     the Individual being evaluated.
     * @return    A long which is the evaluation result.
     * @exception EvaluationException     If there is an evaluation
     *                                    failure.
     */
     public abstract long evaluateLong(Individual i)
            throws EvaluationException;

    /**
     * Evaluate this Gene as a Gene that returns a float when
evaluated.
     *
     * @param     i     the Individual being evaluated.
     * @return    A float which is the evaluation result.
     * @exception EvaluationException     If there is an evaluation
     *                                    failure.
     */
     public abstract float evaluateFloat(Individual i)
            throws EvaluationException;



      /**
       * Evaluate this Gene as a Gene that returns a double when
       * evaluated.
       *
       * @param   i     the Individual being evaluated.
       * @return  A double which is the evaluation result.
       * @exception     EvaluationException     If there is an
       *                                       evaluation failure.
       */
      public abstract double evaluateDouble(Individual i)
            throws EvaluationException;

      /**
       * Evaluate this Gene as a Gene that returns a char when
       * evaluated.
       *
       * @param   i     the Individual being evaluated.
       * @return  A char which is the evaluation result.
       * @exception     EvaluationException     If there is an
       *                                      evaluation failure.
       */
      public abstract char evaluateChar(Individual i)
            throws EvaluationException;

      /**
       * Evaluate this Gene as a Gene that returns a boolean when
       * evaluated.
       *
       * @param   i     the Individual being evaluated.
       * @return  A boolean which is the evaluation result.
       * @exception     EvaluationException     If there is an
       *                                      evaluation failure.
       */
      public abstract boolean evaluateBoolean(Individual i)
            throws EvaluationException;
```

```
        /**
         * Makes a deep clone of this Gene, i.e. the Gene and all of its
         * subtrees.
         *
         * @return  A clone of the Gene and its subtrees.
         */
        abstract public Gene deepClone();

        /**
         * Converts the Gene and its subtrees into a String.
         *
         * @return  A String representing the Gene and any subtrees.
         */
        abstract public String toString();
}//End Gene
```

3. GENEFUNCTION

```
/*
        Copyright Adil Qureshi - 1997
        This code is part of gpsys Release 1.1
        and is released for non-commercial use only.
        Questions, comments etc should be forwarded to :-

                Adil Qureshi
                University College London,
                Department of Computer Science,
                Gower St,
                London WC1E 6BT, UK.
                Tel: +44 (0)171 380 7777 x4436
                Fax: +44 (0)171 387 1397
                email: A.Qureshi@cs.ucl.ac.uk
                URL : http://www.cs.ucl.ac.uk/staff/A.Qureshi/

        Modified by Captain Loretta Vandenberg
                Naval Postgraduate School
                Monterey, CA  93943
*/

package gpsys;

/**
 * A GeneFunction is a Gene tree representing a function call.
 * A GeneFunction therefore has Gene arguments (branches or subtrees).
 *
 * @see     gpsys.Gene
 * @see     gpsys.GeneFunctionGrow
 * @see     gpsys.GeneFunctionFull
 *
 * @version       1.1, 30th June '97
 * @author <a href="mailto:A.Qureshi@cs.ucl.ac.uk">Adil Qureshi</a>
 *       <address>Department of Computer Science,</address>
 *       <address>University College London,</address>
 *       <address>Gower St,</address>
 *       <address>London WC1E 6BT,</address>
 *       <address>UK.</address>
 *
 * @author <a href="mailto:shirley@cs.nps.navy.mil">Rett Vandenberg</a>
 *       <address>Department of Computer Science</address>
 *       <address>Naval Postgraduate School</address>
 *       <address>Monterey, CA  93943</address>
 */
public abstract class GeneFunction extends Gene implements Cloneable {

   /**
    * The arguments for this function call..  The length of this array
is
    * equal to the number of arguments taken by the function referenced
    * by this Gene.
    */
   Gene[] arguments;


   /**
    * Evaluate this Gene as Function returning an Object reference.
    *
    * @param      i     the individual to which this Gene belongs
    * @return     An Object which is the result of the evaluation.
    * @exception  EvaluationException     If there is an evaluation
    *                                     failure.
    *
    */
   public Object evaluateObject(Individual i) throws
      EvaluationException {
            return ((Function) p).evaluateObject(i, arguments);
   }

   /**
    * Evaluate this Gene as Function returning a byte.
    *
    * @param      i     the individual to which this Gene belongs
    * @return     A byte which is the result of the evaluation.
    * @exception  EvaluationException     If there is an evaluation
    *                                     failure.
    *
    */
   public byte evaluateByte(Individual i) throws
      EvaluationException {
            return ((Function) p).evaluateByte(i, arguments);
   }

   /**
    * Evaluate this Gene as Function returning a short.
    *
    * @param      i     the individual to which this Gene belongs
    * @return     A short which is the result of the evaluation.
```

```
  * @exception  EvaluationException     If there is an evaluation
  *                                     failure.
  *
  */
 public short evaluateShort(Individual i) throws
    EvaluationException {
          return ((Function) p).evaluateShort(i, arguments);
 }

 /**
  * Evaluate this Gene as Function returning an int.
  *
  * @param      i     the individual to which this Gene belongs
  * @return     An int which is the result of the evaluation.
  * @exception  EvaluationException     If there is an evaluation
  *                                     failure.
  *
  */
 public int evaluateInt(Individual i) throws
    EvaluationException {
          return ((Function) p).evaluateInt(i, arguments);
 }

/**
 * Evaluate this Gene as Function returning a long.
 *
 * @param i     the individual to which this Gene belongs
 * @return      A long which is the result of the evaluation.
 * @exception   EvaluationException     If there is an
 *                                       evaluation failure.
 *
 */
public long     evaluateLong(Individual i) throws
  EvaluationException {
       return ((Function) p).evaluateLong(i, arguments);
}

/**
 * Evaluate this Gene as Function returning a float.
 *
 * @param i     the individual to which this Gene belongs
 * @return      A float which is the result of the evaluation.
 * @exception   EvaluationException     If there is an
 *                                       evaluation failure.
 *
 */
public float evaluateFloat(Individual i) throws
  EvaluationException {
             return ((Function) p).evaluateFloat(i, arguments);
}

    /**
     * Evaluate this Gene as Function returning a double.
     *
 * @param   i     the individual to which this Gene belongs
 * @return  A double which is the result of the evaluation.
 * @exception     EvaluationException     If there is an
 *                                       evaluation failure.
 *
 */
public double evaluateDouble(Individual i) throws
      EvaluationException {
         return ((Function) p).evaluateDouble(i, arguments);
}

/**
 * Evaluate this Gene as Function returning a char.
 *
 * @param   i     the individual to which this Gene belongs
 * @return  A char which is the result of the evaluation.
 * @exception     EvaluationException     If there is an
 *                                       evaluation failure.
 *
 */
public char evaluateChar(Individual i) throws
      EvaluationException {
         return ((Function) p).evaluateChar(i, arguments);
}


/**
 * Evaluate this Gene as Function returning a boolean.
 *
 * @param   i     the individual to which this Gene belongs
 * @return  A boolean which is the result of the evaluation.
 * @exception     EvaluationException     If there is an
 *                                       evaluation failure.
 *
 */
public boolean evaluateBoolean(Individual i) throws
      EvaluationException {
         return ((Function) p).evaluateBoolean(i, arguments);
}

/**
 * Get the maximum depth of this Gene tree
 *
 * @return  the maximum depth of the tree.
 *
 */
public int depth() {

   // the depth is initially 0
   int d = 0;

   // get the maximum depth of each branch
   for (int i = 0; i < arguments.length; i++) {
      int tmp = arguments[i].depth();
      d = (d < tmp) ? tmp : d;
   }

   // the maximum depth of this Gene is the 1 + the maximum depth
   // of all the branches
   return d + 1;
}

/**
 * Get the number of Genes in this Gene tree.
 *
 * @return  the number of Genes in this Gene tree.
 */
public int complexity() {

   // the total number of Genes is initially one.
   int sum = 1;

   // add the total number of Genes in each argument
   for (int i = 0; i < arguments.length; i++) {
      sum += arguments[i].complexity();
   }
   // the total is this sum
   return sum;
}

/**
 * Added by Rett Vandenberg
 * Get the number of Introns in this Gene tree.
 *
 * @return  the number of Introns in this Gene tree.
 */
public int introns() {

   int sum = 0;

   //if the gene starting the count is an intron
   //the second condition should never have to be used
   //especially if the definition of introns is expanded
   //to other functions besides Nop
   if (this.isIntron() ||
      this.toString().startsWith("(Nop")) {
      sum += 1;
   }

   // add the total number of introns in each argument
   for (int i = 0; i < arguments.length; i++) {
       sum += arguments[i].introns();
   }
   // the total is this sum
   return sum;
}

/**
```

```
   * Added by Rett Vandenberg
   * Get the number of Dominants in this Gene tree.
   *
   * @return  the number of dominants in this Gene tree.
   */
  public int dominants() {

     int sum = 0;

     //if the gene starting the count is a dominant
     if (this.isDominant()) { sum += 1; }

     // add the total number of dominants in each argument
     for (int i = 0; i < arguments.length; i++) {
       sum += arguments[i].dominants();
     }
     // the total is this sum
     return sum;
  }

  /**
   * Added by Rett Vandenberg
   * Get the number of Recessives in this Gene tree.
   *
   * @return  the number of recessives in this Gene tree.
   */
  public int recessives() {

     int sum = 0;

     //if the gene starting the count is a recessive
     if (this.isRecessive()) { sum += 1; }

     // add the total number of introns in each argument
     for (int i = 0; i < arguments.length; i++) {
        sum += arguments[i].recessives();
     }
     // the total is this sum
     return sum;
  }


  /**
   * Make a clone of this Gene tree, cloning all subtrees.
   *
   * @return  a clone of this Gene.
   */
  public Gene deepClone() {
     // the clone is initially empty
     GeneFunction clone = null;
     try {
        // now clone the Gene, and then make arguments refer to
        // clones too
        clone = (GeneFunction) this.clone();
```

```
            clone.p = p.instance();
            clone.arguments = new Gene[arguments.length];
            for (int i = 0; i < arguments.length; i++){
               clone.arguments[i] = (Gene) arguments[i].deepClone();
            }
         }
         catch (CloneNotSupportedException e) {
            // will never happen as long as we implement Cloneable
            // which we do !!
         }
         return clone;
      }

      /**
       * Create a String representing this Gene tree.
       *
       * @return  a String representing this tree.
       */
      public String toString() {
         String s = new String();
         s += "(" + p;
         for (int i = 0; i < arguments.length; i++){
            s += " " + arguments[i];
         }
         s += ")";
         return s;
      }
}//End Gene Function
```

4. GENEFUNCTIONFULL

```
/*
      Copyright Adil Qureshi - 1997
      This code is part of gpsys Release 1.1
      and is released for non-commercial use only.
      Questions, comments etc should be forwarded to :-

            Adil Qureshi
            University College London,
            Department of Computer Science,
            Gower St,
            London WC1E 6BT, UK.
            Tel: +44 (0)171 380 7777 x4436
            Fax: +44 (0)171 387 1397
            email: A.Qureshi@cs.ucl.ac.uk
            URL : http://www.cs.ucl.ac.uk/staff/A.Qureshi/

      Modified by Captain Loretta Vandenberg
            Naval Postgraduate School
            Monterey, CA  93943
*/
```

```
package gpsys;

/**
 * A GeneFunctionFull is a GeneFunction tree created using the Full
 * method.
 * In the Full method, the tree is full depth along any path from the
 * root
 * to a leaf.
 *
 * @version       1.1, 30th June '97
 * @author <a href="mailto:A.Qureshi@cs.ucl.ac.uk">Adil Qureshi</a>
 *        <address>Department of Computer Science,</address>
 *        <address>University College London,</address>
 *        <address>Gower St,</address>
 *        <address>London WC1E 6BT,</address>
 *        <address>UK.</address>
 *
 * @author <a href="mailto:shirley@cs.nps.navy.mil">Rett Vandenberg</a>
 *        <address>Department of Computer Science</address>
 *        <address>Naval Postgraduate School</address>
 *        <address>Monterey, CA  93943</address>
 *
 */
public class GeneFunctionFull extends GeneFunction implements Cloneable
{
  /**
   * Create a Gene tree using the Full method.  The Full method
   * tries to create trees of the maximum specified depth along
   * any branch.
   *
   * @param maxDepth    the maximum depth of the Gene
   *                    tree to be generated.
   * @param type        the required return type of top
   *                    node in the tree.
   * @param gpParameters      the GP parameters for this run.
   * @param adfIndex    the index of the
   *                    ChromosomeParameters to used.
   * @exception   TypeException     If a Function or Terminal of a
   *                    required type could not be
   *                    found.
   *
   */
  public GeneFunctionFull(int maxDepth, Type type,
                        GPParameters gpParameters, int adfIndex)
         throws TypeException {

      // get an array of all functions of the required type
      Function[] functionsOfType = gpParameters.adf[adfIndex].
                            functionsOfTypeAtDepthFull[maxDepth].
                            get(type);

      if (functionsOfType == null) {throw new TypeException(
         "no function of type "+type.toString()
                               +" at depth "+maxDepth);
```

```
}

// then select a function at random from this array
//and assign to p
int code = gpParameters.rng.nextInt()%functionsOfType.length;
code = (code < 0) ? -code : code;

//added by Rett Vandenberg to count the introns
//will have to change if adding different types of introns, but
//this will do for this particular experiment
if (functionsOfType[code].toString() == "Nop"  ) {
   super.intron = true;
   //an intron can never be a dominant gene
   super.dominant = false;
//All recessive functions will be implemented starting with
//an ‘R’.
} else if (functionsOfType[code].toString().startsWith("R")) {
   super.dominant = false;
   //in this particular case, a recessive still adds to the
   //fitness of the individual, so it cannot be considered an
   //intron
   super.intron = false;
} else {//they are dominants
  super.intron = false;
  super.dominant = true;
  //at creation, there is a 50% chance of a double dominant
  //and a 50% chance of a heterozygous dominant
     if((gpParameters.rng.nextInt() % 2) == 0) {
         super.allele = 2; //double dominant
      } else { super.allele = 1; }
   }

   Function f = (Function) functionsOfType[code].instance();
   p = f;

   // now generate the arguments to this function
   arguments = new Gene[f.argTypes.length];
   maxDepth--;
   for (int i = 0; i < arguments.length; i++)
      if (maxDepth > 0) {
        // if there is depth make another GeneFunction
        try {
            arguments[i] = new GeneFunctionFull(maxDepth,
                                                f.argTypes[i],
                                                gpParameters,
                                                adfIndex);
        }
        catch (TypeException e) {
            gpParameters.observer.diagnosticUpdate(
               "GeneFunctionFull " +  e.getMessage());
            arguments[i] = new
            GeneTerminal(maxDepth, f.argTypes[i],
                         gpParameters, adfIndex);
        }
```

```
            } else  // else make a GeneTerminal
                arguments[i] = new GeneTerminal(maxDepth,
                                                f.argTypes[i],
                                                gpParameters,
                                                adfIndex);
    }
}//End GeneFunctionFull
```

5. GENEFUNCTIONGROW

```
/*
      Copyright Adil Qureshi - 1997
      This code is part of gpsys Release 1.1
      and is released for non-commercial use only.
      Questions, comments etc should be forwarded to :-

            Adil Qureshi
            University College London,
            Department of Computer Science,
            Gower St,
            London WC1E 6BT, UK.
            Tel: +44 (0)171 380 7777 x4436
            Fax: +44 (0)171 387 1397
            email: A.Qureshi@cs.ucl.ac.uk
            URL : http://www.cs.ucl.ac.uk/staff/A.Qureshi/

      Modified by Captain Loretta Vandenberg
            Naval Postgraduate School
            Monterey, CA  93943
*/

package gpsys;

/**
 * A GeneFunctionGrow is a GeneFunction created using the Grow method.
 * In the Grow method, the Gene tree is such that, along any branch,
 * the probability of encountering a Terminal is equal to the
 * probability of encountering a Function.
 *
 * @version      1.1, 30th June '97
 * @author <a href="mailto:A.Qureshi@cs.ucl.ac.uk">Adil Qureshi</a>
 *       <address>Department of Computer Science,</address>
 *       <address>University College London,</address>
 *       <address>Gower St,</address>
 *       <address>London WC1E 6BT,</address>
 *       <address>UK.</address>
 *
 * @author <a href="mailto:shirley@cs.nps.navy.mil">Rett Vandenberg</a>
 *       <address>Department of Computer Science</address>
 *       <address>Naval Postgraduate School</address>
 *       <address>Monterey, CA  93943</address>
 *
 */

public class GeneFunctionGrow extends GeneFunction implements Cloneable
{
   /**
    * Create a Gene tree using the Grow method.  The Grow method
    * tries to creates trees in which along any branch, the
    * probability of encountering a Terminal is equal to the
    * probability of enountering a Function.
    *
    * @param      maxDepth    the maximum depth of the Gene
    *                         tree to be generated.
    * @param      type        the required return type of top
    *                   node in the tree.
    * @param      gpParameters      the GP parameters for this run.
    * @param      adfIndex    the index of the
    *                   ChromosomeParameters to used.
    * @exception  TypeException     If a Function or Terminal of
    *                   a required type could not be found.
    *
    */
   public GeneFunctionGrow(int maxDepth, Type type,
                        GPParameters gpParameters, int adfIndex)
            throws TypeException {

      // get an array of all functions of the required type
      Function[] functionsOfType = gpParameters.adf[adfIndex].
                                 functionsOfTypeAtDepthGrow[maxDepth].
                                 get(type);

      if (functionsOfType == null) throw new TypeException(
         "no function of type " + type.toString() + " at depth " +
                                                        maxDepth);

      // then select a function at random from this array and
      // assign to p
      int code =  gpParameters.rng.nextInt()%functionsOfType.length;
      code = (code < 0) ? -code : code;

      //added by Rett Vandenberg to count the introns
      if (functionsOfType[code].toString() == "Nop"  ) {
         super.intron = true;
         super.dominant = false;
      } else if (functionsOfType[code].toString().startsWith("R")) {
         super.dominant = false;
         super.intron = false;
      } else {
         super.intron = false;
         super.dominant = true;
            if((gpParameters.rng.nextInt() % 2) == 0) {
               super.allele = 2; //double dominant
            } else { super.allele = 1; }
      }
      Function f = (Function) functionsOfType[code].instance();
      p = f;

      // now generate the arguments to this function
      arguments = new Gene[f.argTypes.length];
      maxDepth--;
      // just for convenience
      TypeToFunctionsTable typeToFunctions =
gpParameters.adf[adfIndex].
                                   functionsOfTypeAtDepthGrow[maxDepth];

      for (int i = 0; i < arguments.length; i++){

         // depth allowing choose a function or a
         //terminal with a 50% chance for either
            if ((maxDepth > 0) &&
                 ((gpParameters.rng.nextInt() % 2) == 0) &&
               (typeToFunctions.get(f.argTypes[i]) != null)) {
               try {
                  arguments[i] = new GeneFunctionGrow(maxDepth,
                                                       f.argTypes[i],
                                                       gpParameters,
                                                       adfIndex);
               }
               catch(TypeException e) {
                  gpParameters.observer.diagnosticUpdate(
                        "GeneFunctionGrow " + e.getMessage());
                  arguments[i] = new GeneTerminal(maxDepth,
                                                   f.argTypes[i],
                                                   gpParameters,
                                                   adfIndex);
               }
            } else {
                  try {
                     arguments[i] = new GeneTerminal(maxDepth,
                                                      f.argTypes[i],
                                                      gpParameters,
                                                      adfIndex);
                  }
                  catch(TypeException e) {
                     gpParameters.observer.diagnosticUpdate(
                           "GeneFunctionGrow " + e.getMessage());
                     arguments[i] = new GeneFunctionGrow(maxDepth,
                                                          f.argTypes[i],
                                                          gpParameters,
                                                          adfIndex);
                  }
            }
      }
}//End GeneFunctionGrow
```

6. XNO_TYPE

```
/*
      Captain Loretta Vandenberg
*/

package gpsys.primitives;

import gpsys.*;

/**
 * <pre>
 *    NO_TYPE XNo_Type
 * </pre>
 *
 * XNo_Type is a Terminal representing a variable of Type No_Type.
 *
 * @see                   gpsys.Type
 * @see                   gpsys.Primitive
 * @see                   gpsys.Terminal
 *
 * @version     1.1, 26 Aug 99
 * @author  <a href="mailto:shirley@cs.nps.navy.mil">Rett
Vandenberg</a>
 *          <address>Department of Computer Science</address>
 *          <address>Naval Postgraduate School</address>
 *          <address>Monterey, CA  93943</address>
 */
public class XNo_Type extends Terminal {
      /**
       * Holds the value of the variable.
       */
      Type nothing;

      /**
       * Construct a new variable.
       */
      public XNo_Type() {
            this.type = Type.NO_TYPE;
      }

      /**
       * Evaluates the variable, which returns its value.
       *
       * @param   i     The variable being evaluated.
       * @return  A long which is the value of the variable.
       * @exception     EvaluationException     If there is an
       *              evaluation failure.
       */
      public final Type evaluateNo_Type(Individual i) {
      //throws EvaluationException {
         return Type.NO_TYPE;
      }
	/**
	 * Creates a copy of this Terminal.
	 *
	 * @return  Actully a reference to the same Terminal since there
	 *          is no change in instance variables required.
	 */
	public final Primitive instance() {
	   return this;
	}

	/**
	 * Creates a String representing this Terminal.
	 *
	 * @return A String containing the name of this Terminal.
	 */
	public String toString() {
		return "No_Type";
	}
}//End No_Type
```

7. NOP

```
/*
	Loretta Vandenberg
*/

package gpsys.primitives;

import gpsys.*;

/**
 * <pre>
 *    &lt;typeX&gt; Nop()
 * </pre>
 *
 * Nop is a generic Function.  It doesn't do anything,
 * it can just fill a node.
 *
 * @see                gpsys.Type
 * @see                gpsys.Primitive
 * @see                gpsys.Terminal
 *
 * @version     1.1, 22 Aug 99
 * @author  <a href="mailto:shirley@cs.nps.navy.mil">Rett
Vandenberg</a>
 *               <address>Department of Computer Science</address>
 *               <address>Naval Postgraduate School</address>
 *
 */
public class Nop extends Function {

	/**
 * Construct a new Nop Function
 */
public Nop(Type type) {
      this.type = type;
      this.argTypes = new Type[2];
      argTypes[0] = type;
      argTypes[1] = type;
}


/**
* Used to return arguments of FLOAT Types.
*
* @param i              The individual being evaluated.
* @param arguments      The Gene tree representing the arguments.
* @return A float representing the sum of the arguments.
* @exception      EvaluationException     If there is an
*                 evaluation failure.
*/
public final float evaluateFloat(Individual i, Gene[] arguments)
throws EvaluationException {
   return 0.0f;
}


/**
* Used to return arguments of INT Types.
*
* @param i              The individual being evaluated.
* @param arguments      The Gene tree representing the arguments.
* @return A float representing the sum of the arguments.
* @exception      EvaluationException     If there is an
* evaluation failure.
*/
public final int evaluateInt(Individual i, Gene[] arguments)
    throws EvaluationException {
      return 0;
}

//You can add more for the evaluation type needed.  I only used
//these two.

/**
* Creates an instance of this Nop object.  The Type instantiation
* is preserved so that if an Nop is being cloned, the clone will
* also be nop instantiated.
*
* @return   A reference to this Object (yes the same object,
*           since
*                 the type information is to be the same, there
*                 are therefore
*                 no instance variables to be modified.
*/
public final Primitive instance() {
```

```
         return this;
      }

      /**
      * Returns a String representation of this Function.
      *
      * @return A String containing the name of the Function.
      */
      public String toString() {
         return "Nop";
      }
}//End Nop
```

8. RMUL

```
/*
      Captain Loretta Vandenberg
*/

package gpsys.primitives;

import gpsys.*;

/**
 * <pre>
 *    &lt;typeX&gt; RMul(&lt;typeX&gt; num1, &lt;typeX&gt; num2)
 * </pre>
 *
 * RMul is a generic function that returns the result of multiplying
 * both arguments.
 * RMul needs to be Type instantiated during construction to work with
 * particular argument Types.  The supported argument Types include
 * BYTE, SHORT, INT, LONG, FLOAT and DOUBLE.
 *
 * @see                 gpsys.Type
 * @see                 gpsys.Primitive
 * @see                 gpsys.Function
 *
 * @version     1.1, 1 Sep 99
 * @author  <a href="mailto:shirley@cs.nps.navy.mil">Rett
Vandenberg</a>
 *                <address>Department of Computer Science</address>
 *                <address>Naval Postgraduate School</address>
 *                <address>Monterey, CA 93943</address>
 *
 */
public class RMul extends Function {

   /**
    * Constructs a RMul Function that works with the specified Type.
    * The supported Types include BYTE, SHORT, INT, LONG, FLOAT and
    * DOUBLE.
```

```
     *
     * @param type The Type of this RMul Function.
     */
    public RMul(Type type) {
       this.type = type;
       argTypes = new Type[2];
       argTypes[0] = type;
       argTypes[1] = type;
    }

    /**
     * Used to multiply arguments of BYTE Types.
     *
     * @param i                The individual being evaluated.
     * @param arguments  The Gene trees representing the
     *                   arguments to be multiplied.
     * @return A byte representing the product of the arguments.
     * @exception  EvaluationException     If there is an
     * evaluation failure.
     */
    public final byte evaluateByte(Individual i, Gene[] arguments)
       throws EvaluationException {
       return (byte)(arguments[0].evaluateByte(i) *
                                    arguments[1].evaluateByte(i));
    }

    /**
     * Used to multiply arguments of SHORT Types.
     *
     * @param i                The individual being evaluated.
     * @param arguments  The Gene trees representing the
     *           arguments to be multiplied.
     * @return A short representing the product of the arguments.
     * @exception  EvaluationException     If there is an evaluation
     *                                     failure.
     */
    public final short evaluateShort(Individual i, Gene[] arguments)
       throws EvaluationException {
          return (short) (arguments[0].evaluateShort(i) *
                                    arguments[1].evaluateShort(i));
    }

    /**
     * Used to multiply arguments of INT Types.
     *
     * @param i                The individual being evaluated.
     * @param arguments  The Gene trees representing the arguments to be
     *                   added.
     * @return An int representing the product of the arguments.
     * @exception  EvaluationException     If there is an evaluation
     *                                     failure.
     */
    public final int evaluateInt(Individual i, Gene[] arguments)
       throws EvaluationException {
```

```
      return    arguments[0].evaluateInt(i) *
                arguments[1].evaluateInt(i);
}

/**
 * Used to multiply arguments of LONG Types.
 *
 * @param i                The individual being evaluated.
 * @param arguments  The Gene trees representing the arguments to be
 *                   added.
 * @return A long representing the product of the arguments.
 * @exception  EvaluationException     If there is an
 *                                     evaluation failure.
 */
public final long evaluateLong(Individual i, Gene[] arguments)
   throws EvaluationException {
      return    arguments[0].evaluateLong(i) *
                arguments[1].evaluateLong(i);
}

/**
 * Used to multiply arguments of FLOAT Types.
 *
 * @param i                The individual being evaluated.
 * @param arguments  The Gene trees representing the
 *                   arguments to be multiplied.
 * @return A float representing the product of the arguments.
 * @exception  EvaluationException     If there is an
 * evaluation failure.
 */
public final float evaluateFloat(Individual i, Gene[] arguments)
   throws EvaluationException {
      return    arguments[0].evaluateFloat(i) *
                arguments[1].evaluateFloat(i);
}

/**
 * Used to multiply arguments of DOUBLE Types.
 *
 * @param i                The individual being evaluated.
 * @param arguments  The Gene trees representing the arguments to be
 *                   added.
 * @return A double representing the product of the arguments.
 * @exception  EvaluationException     If there is an evaluation
 *                                     failure.
 */
public final double evaluateDouble(Individual i, Gene[] arguments)
   throws EvaluationException {
         return arguments[0].evaluateDouble(i) *
                arguments[1].evaluateDouble(i);
}
```

```
   /**
    * Creates an instance of this Mul object.  The Type instantiation
    * is
    * preserved so that if an INT Mul is being cloned, the clone will
    * also be INT instantiated.
    *
    * @return     A reference to this Object (yes the same object,
    *             since
    *             the type information is to be the same, there are
    *             therefore no instance variables to be modified.
    */
   public final Primitive instance() {
      return this;
   }

   /**
    * Returns a String representation of this Function.
    *
    * @return A String containing the name of the Function.
    */
   public String toString() {
      return "RMul";
   }

}//End RMul
```

9. RADD

```
/*
      Captain Loretta Vandenberg
*/

package gpsys.primitives;

import gpsys.*;

/**
 * <pre>
 *    &lt;typeX&gt; RAdd(&lt;typeX&gt; num1, &lt;typeX&gt; num2)
 * </pre>
 *
 * RAdd is a generic function that returns the result of adding both
 * arguments, but does not perform as efficiently as Add.  It has
 * been added to demonstrate the effect of dominant/recessive
 * genes within a GP.
 * RAdd needs to be Type instantiated during construction to work with
 * particular argument Types.  The supported argument Types include
 * BYTE, SHORT, INT, LONG, FLOAT and DOUBLE.
 *
 * @see                gpsys.Type
 * @see                gpsys.Primitive
 * @see                gpsys.Function
 *
 * @version     1.1, 1 Sep 99
 * @author  <a href="mailto:shirley@cs.nps.navy.mil">Rett
Vandenberg</a>
 *                 <address>Department of Computer Science</address>
 *                 <address>Naval Postgraduate School</address>
 *                 <address>Monterey, CA  93943</address>
 *
 */
public class RAdd extends Function {

   /**
    * Constructs an RAdd Function that works with the specified Type.
    * The supported Types include BYTE, SHORT, INT, LONG, FLOAT and
    * DOUBLE.
    *
    * @param type The Type of this RAdd Function.
    */
   public RAdd(Type type) {
      this.type = type;
      argTypes = new Type[2];
      argTypes[0] = type;
      argTypes[1] = type;
   }

   /**
    * Used to Add arguments of BYTE Types.
    *
    * @param i                  The individual being evaluated.
    * @param arguments  The Gene trees representing the arguments to be
    *                   added.
    * @return A byte representing the sum of the arguments.
    * @exception  EvaluationException     If there is an evaluation
    * failure.
    */
   public final byte evaluateByte(Individual i, Gene[] arguments)
      throws EvaluationException {
            return (byte) (arguments[0].evaluateByte(i) +
                           arguments[1].evaluateByte(i));
   }

   /**
    * Used to Add arguments of SHORT Types.
    *
    * @param i                  The individual being evaluated.
    * @param arguments  The Gene trees representing the arguments to be
    *                   added.
    * @return A short representing the sum of the arguments.
    * @exception  EvaluationException     If there is an evaluation
    *                                     failure.
    */
   public final short evaluateShort(Individual i, Gene[] arguments)
      throws EvaluationException {
         return (short)(arguments[0].evaluateShort(i) +
                     arguments[1].evaluateShort(i));
}

/**
 * Used to Add arguments of INT Types.
 *
 * @param i                The individual being evaluated.
 * @param arguments  The Gene trees representing the arguments to be
 *                   added.
 * @return An int representing the sum of the arguments.
 * @exception  EvaluationException     If there is an evaluation
 *                                     failure.
 */
public final int evaluateInt(Individual i, Gene[] arguments)
   throws EvaluationException {
      return    arguments[0].evaluateInt(i) +
                arguments[1].evaluateInt(i);
}

/**
 * Used to Add arguments of LONG Types.
 *
 * @param i                The individual being evaluated.
 * @param arguments  The Gene trees representing the arguments to be
 *                  added.
 * @return A long representing the sum of the arguments.
 * @exception  EvaluationException     If there is an evaluation
 *                                     failure.
 */
public final long evaluateLong(Individual i, Gene[] arguments)
   throws EvaluationException {
      return    arguments[0].evaluateLong(i) +
                arguments[1].evaluateLong(i);
}

/**
 * Used to Add arguments of FLOAT Types.
 *
 * @param i                The individual being evaluated.
 * @param arguments  The Gene trees representing the arguments to be
 *                   added.
 * @return A float representing the sum of the arguments.
 * @exception  EvaluationException     If there is an evaluation
 *                                     failure.
 */
public final float evaluateFloat(Individual i, Gene[] arguments)
   throws EvaluationException {
      return    arguments[0].evaluateFloat(i) +
                arguments[1].evaluateFloat(i);
}

/**
 * Used to Add arguments of DOUBLE Types.
 *
```

```
     * @param i                  The individual being evaluated.
     * @param arguments  The Gene trees representing the arguments to be
     *                   added.
     * @return A double representing the sum of the arguments.
     * @exception  EvaluationException     If there is an evaluation
     *                                     failure.
     */
    public final double evaluateDouble(Individual i, Gene[] arguments)
       throws EvaluationException {
          return    arguments[0].evaluateDouble(i) +
                    arguments[1].evaluateDouble(i);
    }

    /**
     * Creates an instance of this RAdd object.  The Type instantiation
     * is preserved so that if an INT RAdd is being cloned, the clone
     * will also be INT instantiated.
     *
     * @return     A reference to this Object (yes the same object,
since
     *             the type information is to be the same, there are
     *             therefore
     *             no instance variables to be modified.
     */
    public final Primitive instance() { return this; }

    /**
     * Returns a String representation of this Function.
     *
     * @return A String containing the name of the Function.
     */
    public String toString() { return "RAdd"; }

}//End RAdd
```

10. GEOSERIES

```
/*
     Loretta Vandenberg
*/

package gpsys.series;
import gpsys.*;
import java.io.*;

/**
 * The user interface for the geometric series evaluation.  This class
 * also contains the main() function to execute the application.
 *
 * @version     1.1, 30th June '97
 * @author  <a href="mailto:shirley@cs.nps.navy.mil">Rett
Vandenberg</a>
 *                <address>Department of Computer Science,</address>
 *                 <address>Naval Postgraduate School</address>
 *                 <address>Monterey, CA  93943</address>
 *
 **/
public class GeoSeries implements GPObserver {
   /**
    * The filePrefix to use when saving reports and generation states
    * during
    * evolution.
    */
   String filePrefix;

   /**
    * Construct the geometric series expansion user interface using the
    * specified file prefix.
    *
    * @param filePrefix The file prefix to be used for saving reports
    * and generation states.
    */
   public GeoSeries(String filePrefix) {this.filePrefix = filePrefix;}

   /**
    * If the filePrefix is null, just write a report of the current
    * generation to the standard output.  Otherwise, also append the
    * report to the file "filePrefix.txt" and save the current
    * generation to the file "filePrefix.p1.gzip".
    *
    * @param gpParameters     The GP parameters used for this run.
    * @param how              How the generation was created.  Can be
    *                         either
    *                      CREATION, FROMSTREAM or EVOLVED.
    */
   public void generationUpdate(GPParameters gpParameters, int how) {
      // if a file prefix was given, write the report to file and save
      // the current generation.  This need not be done if the
      // generation was just loaded from file.
      if ((filePrefix != null) && (how == GPObserver.CREATION ||
           how == GPObserver.EVOLVED)) {
         // try to save the current generation
         try {
            diagnosticUpdate("Saving current generation...");
            gpParameters.save(filePrefix);
            diagnosticUpdate("Saved  current generation.");
         }
         catch (IOException e) {
            System.out.println("gpParameters.save() : " + e);
            System.exit(1);
         }

        // try to write a report of the current population to file
        try {
           gpParameters.writeReport(filePrefix,how ==
                                                GPObserver.CREATION);
        }
```

```
         catch (IOException e) {
            System.out.println("gpParameters.writeReport() : " + e);
            System.exit(1);
         }
      }

      // now write a report of the current population to the standard
      // output
      StringWriter sw = new StringWriter();
      PrintWriter pw = new PrintWriter(sw);
      gpParameters.writeReport(pw,(how == GPObserver.CREATION) ||
                               (how == GPObserver.FROMSTREAM));
      System.out.print(sw.toString());
   }

   /**
    * We are not interested in this update, so we just ignore it.
    *
    * @param      gpParameters      the GP parameters used for this
run.
    * @param      i           the Individual that has just been
created.
    * @param      creationIndex     the index of the Individual in the
    *                               population.
    *
    */
   public void individualUpdate(GPParameters gpParameters,
                             Individual i, int creationIndex) {}

   /**
    * We are not interested in this update, so we just ignore it.
    *
    * @param      gpParameters      the GP parameters used for this
run.
    * @param      individualIndex   the index of the created Individual
    *                               in the population.
    * @param      creationMethod    how the Individual was created.
Can
    *                               be either VIA_MUTATION or
    *                               VIA_CROSSOVER.
    */
   public void individualUpdate(GPParameters gpParameters,
      int individualIndex, int creationIndex) {}


   /**
    * Print the diagnostic message to the standard output.
    *
    * @param s    The diagnostic message.
    */
   public void diagnosticUpdate(String s) { System.out.println(s); }

   /**
    * Print the exception and generate a stack trace on the standard
```

```
     * output.
     *
     * @param e    The exception that was genearted.
     */
   public void exception(GPException e) {
      System.out.println(e.getMessage());
      e.printStackTrace();
      System.exit(0);
   }

  /**
   * The main() method of the geometric series application.  This
   * application may be invoked in any of the following ways.
   * <pre>
   *         java GeoSeries &lt;filePrefix&gt;
   *         java GeoSeries &lt;filePrefix&gt; &lt;generations>
   *         java GeoSeries &lt;filePrefix&gt; &lt;RNG seed>
   *                        &lt;population> &lt;generations>
   *         java GeoSeries &lt;RNG seed&gt; &lt;population>
   *                        &lt;generations>
   * </pre>
   *
   * The first instructs the application to restart from the last saved
   * session using the files with the prefix specified.  The second is
   * the same as the first, except that the maximum number of
   * generations is modified as specified.  The third istructs the
   * application to start a new run using the specifed file prefix for
   * saves, and the specified parameters for the run.  The last is the
   * similar to the previous invocation except that nothing is saved to
   * disk and is as a result very fast.
   */
   public static void main(String[] argv) {

      GeoSeriesGPParameters gpParameters = null;
      String filePrefix = null;

      // now read the command line arguments
      switch(argv.length) {
         case 1: // load entirely from saved session
         case 2: // load from saved session but change the max
                 // generations
            filePrefix = argv[0];
            try {
               System.out.println("Loading last saved generation...");
               gpParameters = (GeoSeriesGPParameters)
                                    GPParameters.load(filePrefix);
               if (gpParameters != null)
                  System.out.println("Loaded last saved
generation...");
            }
            catch (java.io.IOException e) {
               System.out.println("Loading problem : " + e);
               System.exit(1);
            }
            catch (ClassNotFoundException e) {
               System.out.println("Loading problem : " + e);
               System.exit(1);
            }
            if (argv.length == 2) {
               int generations = Integer.parseInt(argv[1]);
               gpParameters.generations = generations;
            }
            break;
         case 3: {
            long rngSeed = Long.parseLong(argv[0]);
            int population = Integer.parseInt(argv[1]);
            int generations   = Integer.parseInt(argv[2]);
            gpParameters = new GeoSeriesGPParameters(rngSeed,
                                     population, generations);
            }
            break;
         case 4: {
            filePrefix = argv[0];
            long rngSeed = Long.parseLong(argv[1]);
            int population = Integer.parseInt(argv[2]);
            int generations = Integer.parseInt(argv[3]);
            gpParameters = new GeoSeriesGPParameters(rngSeed,
                                     population, generations);
            }
            break;
         default:
            System.out.println("Usage : GeoSeries <file>");
            System.out.println("        GeoSeries <file>
                                           <generations>");
            System.out.println("        GeoSeries " +
                  "<RNG seed> <population> <generations>");
            System.out.println("        GeoSeries " +
               "<file> <RNG seed> <population> <generations>");
            return;
         }

         // set the observer to be an instance of our user interface
         gpParameters.observer = new GeoSeries(filePrefix);

         // create a new GP system
         GPsys gpSys = new GPsys(gpParameters);
         // and start evolving !!!
         gpSys.evolve();
      }
}//End GeoSeries
```

11. GEOSERIESGPPPARAMETERS

```
/*
     Loretta Vandenberg
*/

package gpsys.series;

import gpsys.*;
import java.util.Random;

/**
 * The GPParameters class for the geometric series problem.  Any
 * instance of this class has all the GP parameters correctly set.
 *
 * @version     1.1, 17 Aug 99
 * @author  <a href="mailto:shirley@cs.nps.navy.mil">Rett
Vandenberg</a>
 *                <address>Department of Computer Science,</address>
 *                <address>Naval Postgraduate School</address>
 *                <address>Monterey, CA  93943</address>
 **/
public class GeoSeriesGPParameters extends GPParameters {

   /**
    * Constructs a GPParameters object for the geometric series
    * expansion problem.
    *
    * @param seed The seed to be used for the random number generator.
    *             If a seed value of 0 is supplied, a unique seed is
    *             generated using the current time.
    * @param population The size of the population to be used.
    * @param generations  The maximum number of generations to be
    *                     evolved.
    */
   GeoSeriesGPParameters(long seed, int population, int generations ) {

      this.populationSize = population;
      this.generations = generations;

      // create a seed using the current time if the seed suppied is 0
      if (seed == 0){rngSeed = System.currentTimeMillis();}
      else {rngSeed = seed;}
      rng = new Random(rngSeed);

      //this has to be modified for each run if wanting to change
      pMutation   = 0.0;
      tournamentSize = 7;

      adf = new ChromosomeParameters[1];
      adf[0] = new GeoSeriesChromosomeParametersADF0(rng);
      fitness = new GeoSeriesFitness();
   }
}//End GeoSeriesGPParameters
```

12. GEOSERIESFITNESS

```
/*
      Loretta Vandenberg
*/

package gpsys.series;

import gpsys.*;
import gpsys.primitives.*;

/**
 * The Fitness class for the series expansion problem.  The fitness is
 * measured by the absolute value of the difference between the real
 * function and the function generated by the GP system.  The fitness
of
 * one GP is is deemed better than another if either the error is
 * smaller or if the error is equal, but the complexity is smaller.
The
 * termination criteria is met when the error is zero.  This problem
 * involves finding the function :-
 * <pre>
 *                 x^y + x^(y-1) + x^(y-2)… + x^2 + x + 1
 * </pre>
 *
 * @version     1.1, 15 Aug 99
 * @author  <a href="mailto:shirley@cs.nps.navy.mil">Rett
Vandenberg</a>
 *                 <address>Department of Computer Science,</address>
 *                 <address>Naval Postgraduate School</address>
 *                 <address>Monterey, CA  93943</address>
 **/
public class GeoSeriesFitness extends Fitness {
   /**
    * The error between the ideal function and the GP.
    */
   double fitness;

   /**
    * The complexity of the GP i.e. the number of Genes it contains.
    */
   int complexity;

   /**
    * The minimum value of x to be used for testing the evolved
    * function.
    */
   public static float from = 0.0;

   /**
    * The maximum value of x to be used for testing the evolved
    * function.
    */
   public static float to = 10.0;

   /**
    * The number of samples used to test the evolved function.
    */
   public static int samples = 50;

      /**
       * Construct a Fitness object with default Fitness.
       */
      public GeoSeriesFitness() {
            fitness = 0.0;
            complexity = 0;
      }

      /**
       * Constructs a Fitness object by evaluating an Individual.
       *
       * @param   gpParameters      The GP parameters for this run.
       * @param   i           The individual to be evaluated.
       */
      public GeoSeriesFitness(GPParameters gpParameters, Individual i)
{

         complexity = i.complexity();
         // calculate the raw fitness

         fitness = 0.0;
         float step = (to - from) / samples;

         // xFloat is reference to the XFloat Terminal used to
         // generate ADF0.
         XFloat xFloat = (XFloat)

i.adf[0].gpParameters.adf[0].terminals[0];

         for (float j = from; j <= to; j += step) {
            xFloat.set(j);
            float guess = 0.0f;

            int power = gpParameters.rng.nextInt();
            float real;

            for (int pow = power; pow >=0; pow--) {
               real += pow(j,power);
            }
            //add the 1
            real += 1.0;

            try {
               guess = i.evaluateFloat();
               fitness += Math.abs(real - guess);
      }
      catch (DivideByZeroException e) {
         // a divide by zero error is really bad
         // so don't bother with the remaining tests
         fitness = Float.MAX_VALUE;
         break;
      }
      catch (GPException e) {
         gpParameters.observer.exception(e);
      }
   }


   //subtract from fitness for recessives
   //in this case, one would add
   //fitness += i.recessives;
}

/**
 * Adds a fitness value to this fitness value.
 *
 * @param f the fitness value to be added.
 */
public void add(Fitness f) {
      fitness += ((GeoSeriesFitness)f).fitness;
      complexity += ((GeoSeriesFitness)f).complexity;
      recessives +=((GeoSeriesFitness)f).recessives;
};

/**
 * Added by Rett Vandenberg
 * Subtracts a fitness value to this fitness value.
 *
 * @ param f      the fitness value to be subtracted.
 */
public void subtract(Fitness f) {
      fitness -= ((GeoSeriesFitness)f).fitness;
      complexity += ((GeoSeriesFitness)f).complexity;
      recessives += ((GeoSeriesFitness)f).recessives;
}

/**
 * Divide the fitness by the specified integer.  This is used by
 * the GP system to calculate the average fitness of the
 * population.
 *
 * @param   divisor     the integer to divide the fitness by.
 */
public void divide(int divisor) {
      fitness /= divisor;
      complexity  /= divisor;
}

/**
```

```
         * Tests if this fitness value is greater than another fitness
         * value.
         *
         * @param   f     the fitness with which to compare.
         * @return  true if f has higher fitness, false otherwise.
         */
        public boolean greaterThan(Fitness f) {
                if (fitness < ((GeoSeriesFitness)f).fitness)
                        return true;
                if (fitness == ((GeoSeriesFitness)f).fitness)
                        return complexity  <
((GeoSeriesFitness)f).complexity;
                return false;
        }

        /**
         * Tests if this fitness is less than another fitness.
         *
         * @param   f     the fitness with which to compare.
         * @return  true if f has less fitness, false otherwise.
         */
        public boolean lessThan(Fitness f) {
                if (fitness > ((GeoSeriesFitness)f).fitness)
                        return true;
                if (fitness == ((GeoSeriesFitness)f).fitness)
                        return complexity  >
((GeoSeriesFitness)f).complexity;
                return false;
        }

        /**
         * Tests if this fitness is equal to another fitness.
         *
         * @param   f     the fitness with which to compare.
         * @return  true if f has the same fitness, false otherwise.
         */
        public boolean equals(Fitness f) {
                return
                        (fitness == ((GeoSeriesFitness)f).fitness) &&
                        (complexity  == ((GeoSeriesFitness)f).complexity);
        }

        /**
         * Creates a new instance of the Fitness object with a default
         * fitness.
         *
         * @return  an instance of a Fitness object with default fitness.
         */
        public Fitness instance() {
                return new GeoSeriesFitness();
        }

        /**
         * Creates a new instance of the Fitness object which represents
```

```
	 * the fitness of the specified individual.
	 *
	 * @param   gpParameters      the parameters for this GP run.
	 * @param   i                 the individual to be evaluated.
	 * @return  The Fitness of the specified individual.
	 */
	public Fitness instance(GPParameters gpParameters, Individual i)
{
		return new GeoSeriesFitness(gpParameters, i);
	}

	/**
	 * Tests whether this fitness meets the termination criteria.
	 *
	 * @return  true if the termination criteria has been met, false
	 *          otherwise.
	 */
	public boolean terminationCondition() {return fitness == 0;}


	/**
	 * Converts the fitness into a String suitable for printing.
	 *
	 * @return  A String representing the fitness.
	 */
	public String toString() {
		return fitness + ",";// + complexity ;
	       //when testing
	       //return "Fitness( " + fitness + "," + complexity + ")";
	}
}//End GeoSeriesFitness
```

13. GEOSERIESCHROMOSOMEPARAMETERSADF0

```
/*
	Loretta Vandenberg
*/

package gpsys.series;

import java.util.Random;
import gpsys.*;
import gpsys.primitives.*;

/**
 * The Chromosome parameters for ADF0 (the result producing branch) of
 * the series expansion problem.
 *
 * ADF0 has the following Function and Terminal sets.
 *
 * <pre>
 *	Functions = {Add, Mult, RAdd, RMul, Nop}
 *          Terminals = {X, 1}
 * </pre>
 *
 * @version     1.1, 30th June '97
 * @author  <a href="mailto:shirley@cs.nps.navy.mil">Rett
Vandenberg</a>
 *          <address>Department of Computer Science,</address>
 *          <address>Naval Postgraduate School</address>
 *          <address>Monterey, CA  93943</address>
 */
public class GeoSeriesChromosomeParametersADF0 extends
   ChromosomeParameters {

   /**
    * Create the ChromosomeParameters object for ADF0.
    */
   public GeoSeriesChromosomeParametersADF0(Random rng) {

      maxDepth = 9;
      maxDepthAtCreation = 7;
      maxDepthMutation  = 3;

      // the return type of the Chromosome when it is evaluated
      type = Type.FLOAT;

      // the types used by this chromosome
      types = new Type[2];
      types[0] = Type.FLOAT;
      types[1] = Type.BOOLEAN;
      //types[2] = Type.NO_TYPE;

      //define the function set
      functions = new Function[5];
      functions[0] = new Add(Type.FLOAT);
      functions[1] = new Mul(Type.FLOAT);
      functions[2] = new RAdd(Type.FLOAT);
      functions[3] = new RMul(Type.FLOAT);
      functions[4] = new Nop(Type.FLOAT);


      // define the terminal set
      terminals = new Terminal[2];
      terminals[0] = new XFloat();
      terminals[1] = new One(Type.FLOAT);
      //terminals[2] = new XNo_Type();

      createMethod = CREATE_RAMP_HALF_AND_HALF;
   }

}//GeoSeriesChromosomeParametersADF0
```

LIST OF REFERENCES

1. Fogel, D. B., *Evolutionary Computation: Toward a New Philosophy of Machine Intelligence*, p. 6, IEEE Press, 1995.
2. Holland, J. H., *Adaptation in Natural and Artificial Systems*, The MIT Press, 1992.
3. Qureshi, A. "GPsys 1.1." [http://www.cs.ucl.uk/staff/A.Qureshi/gpsys.html]. June 1997.
4. Watson, J. D., and others, *Molecular Biology of the Gene*, 4th ed., vol. 1, p. 4, Benjamin/Cummings Publishing Company, Inc., 1987.
5. Haldane, J. B. S., *The Causes of Evolution*, p. 1, Princeton University Press, 1990.
6. Watson, J. D., and others, *Molecular Biology of the Gene*, 4th ed., vol. 1, p. 8, Benjamin/Cummings Publishing Company, Inc., 1987.
7. Stryer, L., *Biochemistry*, 3d ed., p. 74, W. H. Freeman and Company, 1988.
8. Stryer, L., *Biochemistry*, 3d ed., p. 112, W. H. Freeman and Company, 1988.
9. Stryer, L., *Biochemistry*, 3d ed., p. 112, W. H. Freeman and Company, 1988.
10. Mitchell, M., *An Introduction to Genetic Algorithms*, p. 3, The MIT Press, 1996.
11. Koza, J. R., *Genetic Programming: On the Programming of Computers by Natural Selection*, The MIT Press, 1992.
12. Bolt, Beranek and Newman, Inc Report 7866, *Strongly Typed Genetic Programming*, D. J. Montana, p. 5, 25 March 1994.
13. Nordin, P., "A Compiling Genetic Programming System that Directly Manipulates the Machine Code" in *Advances in Genetic Programming*, pp. 311-331, K. E. Kinnear, Jr. ed., The MIT Press, 1994.
14. Crepeau, R. L. "Genetic Evolution of Machine Language Software" in *Genetic Programming: From Theory to Real-World Applications. Proceedings of the Workshop Organized in Conjunction with the Twelfth International Conference on Machine Learning*, pp. 121-135, J. P. Rosca ed., University of Rochester Press, 1995.

15. Crepeau, R. L. "Genetic Evolution of Machine Language Software" in *Genetic Programming: From Theory to Real-World Applications. Proceedings of the Workshop Organized in Conjunction with the Twelfth International Conference on Machine Learning*, pp. 126, J. P. Rosca ed., University of Rochester Press, 1995.
16. Greene, F. "A Method for Utilizing Diploid/Dominance in Genetic Search" in *Proceedings of the First IEEE International Conference on Evolutionary Computation*, pp. 439-444, Z. Michalewicz, and others eds., IEEE Press, 1994.
17. Whitney, D., "The Genitor Group." [http://www.cs.colostate.edu/~genitor/]. July 1999.
18. Nordin, P., Francone F. and Banzhaf, W., "Explicitly Defined Introns and Destructive Crossover in Genetic Programming" in *Advances in Genetic Programming*, vol. 2, pp. 111-134, P. Angeline and K. E. Kinnear, Jr. eds., The MIT Press, 1996.
19. Qureshi, A. "GPsys 1.1." [http://www.cs.ucl.uk/staff/A.Qureshi/gpsys.html]. June 1997.
20. Singleton, A. "GP-Quick: A Simple Genetic Programming System in C++." [http://www.cwi.nl/ftp/W.B.Langdon/gp-code/simple/]. April 1997.
21. Wall, M. B., "GALib: A C++ Genetic Algorithms Library." [http://lancet.mit.edu/galib-2.4/GAlib.html]. May 1996.
22. Punch, W. and Goodman E., "Genetic Algorithms Research and Applications Group (GARAGe)." [http://GARAGe.cps.msu.edu]. April 1999.
23. Whitney, D., "The Genitor Group." [http://www.cs.colostate.edu/~genitor/]. July 1999.
24. Adami, C., *Introduction to Artificial Life*, Telos Springer-Verlag, Inc., 1998.
25. University of Salford, Cybernetics Research Institute Report 040, *Genetic Programming in C++*, by A. P. Fraser, 1994.
26. Stryer, L., *Biochemistry*, 3d ed., p. 676, W. H. Freeman and Company, 1988.
27. Stryer, L., *Biochemistry*, 3d ed., p. 676, W. H. Freeman and Company, 1988.
28. Stryer, L., *Biochemistry*, 3d ed., pp. 113 and 725, W. H. Freeman and Company, 1988.
29. Watson, J. D., and others, *Molecular Biology of the Gene*, 4th ed., vol. 1, p. 459, Benjamin/Cummings Publishing Company, Inc., 1987.

30. Watson, J. D., and others, *Molecular Biology of the Gene*, 4th ed., vol. 1, p. 459, Benjamin/Cummings Publishing Company, Inc., 1987.
31. Watson, J. D., and others, *Molecular Biology of the Gene*, 4th ed., vol. 1, p. 459, Benjamin/Cummings Publishing Company, Inc., 1987.
32. Gillespie, J. H., *Population Genetics: A Concise Guide*, pp. 19-47, The John Hopkins University Press, 1998.

INITIAL DISTRIBUTION LIST

1. Defense Technical Information Center 2
   8725 John J. Kingman Road, STE 0944
   Ft. Belvoir, Virginia 22060-6128

2. Dudley Knox Library 2
   Naval Postgraduate School
   411 Dyer Road
   Monterey, California 93943-5101

3. Director, Training and Education.. 1
   MCCDC, Code C46
   1019 Elliot Road
   Quantico, Virginia 22134-5027

4. Director, Marine Corps Research Center 2
   MCCDC, Code C40RC
   2040 Broadway Street
   Quantico, Virginia 22134-5107

5. Director, Studies and Analysis Division.. 1
   MCCDC, Code C45
   300 Russell Road
   Quantico, Virginia 22134-5130

6. Professor Taylor Kidd, Code CS/KD.. 1
   Naval Postgraduate School
   Monterey, California 93943

7. Professor Craig Rasmussen, Code MA/RS.. 1
   Naval Postgraduate School
   Monterey, California 93943

8. Capt Loretta L. Vandenberg.. 2
   804 Hogan Dr.
   Las Vegas, Nevada 89107

9. Chairman, Code CS.. 1
   Naval Postgraduate School
   Monterey, California 93943